Table of Contents

1. About the Firebird 3.0 Language Reference .......................................................... 15
   1.1. Subject .................................................................................................................. 15
   1.2. Authorship .......................................................................................................... 15
       1.2.1. Contributors ............................................................................................... 15
   1.3. Acknowledgments ............................................................................................... 16
   1.4. Contributing ........................................................................................................ 16

2. SQL Language Structure .......................................................................................... 17
   2.1. Background to Firebird’s SQL Language .............................................................. 17
       2.1.1. SQL Flavours .............................................................................................. 17
       2.1.2. SQL Dialects .............................................................................................. 17
       2.1.3. Error Conditions ........................................................................................ 19
   2.2. Basic Elements: Statements, Clauses, Keywords .................................................. 19
   2.3. Identifiers ............................................................................................................ 20
       2.3.1. Rules for Regular Object Identifiers ............................................................ 20
       2.3.2. Rules for Delimited Object Identifiers ........................................................ 20
   2.4. Literals .................................................................................................................. 21
   2.5. Operators and Special Characters ...................................................................... 21
   2.6. Comments ........................................................................................................... 22

3. Data Types and Subtypes ......................................................................................... 24
   3.1. Integer Data Types .............................................................................................. 26
       3.1.1. SMALLINT ................................................................................................. 26
       3.1.2. INTEGER ..................................................................................................... 26
       3.1.3. BIGINT ....................................................................................................... 27
       3.1.4. Hexadecimal Format for Integer Numbers ................................................. 27
   3.2. Floating-Point Data Types ................................................................................. 28
       3.2.1. FLOAT ......................................................................................................... 28
       3.2.2. DOUBLE PRECISION .............................................................................. 28
   3.3. Fixed-Point Data Types ...................................................................................... 28
       3.3.1. NUMERIC ................................................................................................. 29
       3.3.2. DECIMAL ................................................................................................. 30
   3.4. Data Types for Dates and Times ......................................................................... 30
       3.4.1. DATE .......................................................................................................... 31
       3.4.2. TIME ......................................................................................................... 31
       3.4.3. TIMESTAMP ............................................................................................... 32
       3.4.4. Operations Using Date and Time Values .................................................... 32
   3.5. Character Data Types .......................................................................................... 33
       3.5.1. Unicode ....................................................................................................... 33
       3.5.2. Client Character Set .................................................................................... 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.3. Special Character Sets</td>
<td>34</td>
</tr>
<tr>
<td>3.5.4. Collation Sequence</td>
<td>34</td>
</tr>
<tr>
<td>3.5.5. Character Indexes</td>
<td>35</td>
</tr>
<tr>
<td>3.5.6. Character Types in Detail</td>
<td>36</td>
</tr>
<tr>
<td>3.6. Boolean Data Type</td>
<td>38</td>
</tr>
<tr>
<td>3.6.1. BOOLEAN</td>
<td>38</td>
</tr>
<tr>
<td>3.7. Binary Data Types</td>
<td>40</td>
</tr>
<tr>
<td>3.7.1. BLOB Subtypes</td>
<td>40</td>
</tr>
<tr>
<td>3.7.2. BLOB Specifics</td>
<td>41</td>
</tr>
<tr>
<td>3.7.3. ARRAY Type</td>
<td>42</td>
</tr>
<tr>
<td>3.8. Special Data Types</td>
<td>44</td>
</tr>
<tr>
<td>3.8.1. SQL_NULL Data Type</td>
<td>44</td>
</tr>
<tr>
<td>3.9. Conversion of Data Types</td>
<td>45</td>
</tr>
<tr>
<td>3.9.1. Explicit Data Type Conversion</td>
<td>45</td>
</tr>
<tr>
<td>3.9.2. Implicit Data Type Conversion</td>
<td>49</td>
</tr>
<tr>
<td>3.10. Custom Data Types — Domains</td>
<td>51</td>
</tr>
<tr>
<td>3.10.1. Domain Attributes</td>
<td>51</td>
</tr>
<tr>
<td>3.10.2. Domain Override</td>
<td>51</td>
</tr>
<tr>
<td>3.10.3. Creating and Administering Domains</td>
<td>52</td>
</tr>
<tr>
<td>3.11. Data Type Declaration Syntax</td>
<td>53</td>
</tr>
<tr>
<td>3.11.1. Scalar Data Types Syntax</td>
<td>53</td>
</tr>
<tr>
<td>3.11.2. BLOB Data Types Syntax</td>
<td>55</td>
</tr>
<tr>
<td>3.11.3. Array Data Types Syntax</td>
<td>55</td>
</tr>
<tr>
<td>4. Common Language Elements</td>
<td>57</td>
</tr>
<tr>
<td>4.1. Expressions</td>
<td>57</td>
</tr>
<tr>
<td>4.1.1. Literals (Constants)</td>
<td>59</td>
</tr>
<tr>
<td>4.1.2. SQL Operators</td>
<td>62</td>
</tr>
<tr>
<td>4.1.3. Conditional Expressions</td>
<td>65</td>
</tr>
<tr>
<td>4.1.4. NULL in Expressions</td>
<td>67</td>
</tr>
<tr>
<td>4.1.5. Subqueries</td>
<td>68</td>
</tr>
<tr>
<td>4.2. Predicates</td>
<td>69</td>
</tr>
<tr>
<td>4.2.1. Conditions</td>
<td>70</td>
</tr>
<tr>
<td>4.2.2. Comparison Predicates</td>
<td>70</td>
</tr>
<tr>
<td>4.2.3. Existential Predicates</td>
<td>83</td>
</tr>
<tr>
<td>4.2.4. Quantified Subquery Predicates</td>
<td>86</td>
</tr>
<tr>
<td>5. Data Definition (DDL) Statements</td>
<td>89</td>
</tr>
<tr>
<td>5.1. DATABASE</td>
<td>89</td>
</tr>
<tr>
<td>5.1.1. CREATE DATABASE</td>
<td>89</td>
</tr>
<tr>
<td>5.1.2. ALTER DATABASE</td>
<td>95</td>
</tr>
<tr>
<td>5.1.3. DROP DATABASE</td>
<td>100</td>
</tr>
<tr>
<td>5.2. SHADOW</td>
<td>101</td>
</tr>
</tbody>
</table>
**Table of Contents**

5.2.1. CREATE SHADOW ................................................. 101  
5.2.2. DROP SHADOW ................................................. 103  

5.3. DOMAIN ........................................................... 104  
5.3.1. CREATE DOMAIN .............................................. 104  
5.3.2. ALTER DOMAIN ............................................ 109  
5.3.3. DROP DOMAIN ............................................... 112  

5.4. TABLE .............................................................. 113  
5.4.1. CREATE TABLE .............................................. 113  
5.4.2. ALTER TABLE ............................................. 129  
5.4.3. DROP TABLE ................................................ 135  
5.4.4. RECREATE TABLE ........................................... 136  

5.5. INDEX .............................................................. 137  
5.5.1. CREATE INDEX ............................................ 137  
5.5.2. ALTER INDEX ............................................. 140  
5.5.3. DROP INDEX ................................................. 142  
5.5.4. SET STATISTICS ........................................... 143  

5.6. VIEW ............................................................... 144  
5.6.1. CREATE VIEW ............................................... 144  
5.6.2. ALTER VIEW ............................................... 148  
5.6.3. CREATE OR ALTER VIEW ................................... 149  
5.6.4. DROP VIEW ................................................ 150  
5.6.5. RECREATE VIEW ............................................ 151  

5.7. TRIGGER .......................................................... 152  
5.7.1. CREATE TRIGGER ........................................... 152  
5.7.2. ALTER TRIGGER ............................................ 164  
5.7.3. CREATE OR ALTER TRIGGER ................................ 166  
5.7.4. DROP TRIGGER ............................................... 167  
5.7.5. RECREATE TRIGGER ........................................ 168  

5.8. PROCEDURE ........................................................ 168  
5.8.1. CREATE PROCEDURE ......................................... 169  
5.8.2. ALTER PROCEDURE .......................................... 173  
5.8.3. CREATE OR ALTER PROCEDURE ...................................... 174  
5.8.4. DROP PROCEDURE ........................................... 175  
5.8.5. RECREATE PROCEDURE ....................................... 176  

5.9. FUNCTION ........................................................ 177  
5.9.1. CREATE FUNCTION ........................................... 177  
5.9.2. ALTER FUNCTION ............................................ 183  
5.9.3. CREATE OR ALTER FUNCTION ................................... 184  
5.9.4. DROP FUNCTION ............................................... 185  
5.9.5. RECREATE FUNCTION ....................................... 186  

5.10. EXTERNAL FUNCTION ........................................... 187
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.4. Joins</td>
<td>244</td>
</tr>
<tr>
<td>6.1.5. The WHERE clause</td>
<td>254</td>
</tr>
<tr>
<td>6.1.6. The GROUP BY clause</td>
<td>257</td>
</tr>
<tr>
<td>6.1.7. The PLAN clause</td>
<td>262</td>
</tr>
<tr>
<td>6.1.8. UNION</td>
<td>270</td>
</tr>
<tr>
<td>6.1.9. ORDER BY</td>
<td>272</td>
</tr>
<tr>
<td>6.1.10. ROWS</td>
<td>275</td>
</tr>
<tr>
<td>6.1.11. OFFSET, FETCH</td>
<td>279</td>
</tr>
<tr>
<td>6.1.12. FOR UPDATE [OF]</td>
<td>281</td>
</tr>
<tr>
<td>6.1.13. WITH LOCK</td>
<td>281</td>
</tr>
<tr>
<td>6.1.14. INTO</td>
<td>285</td>
</tr>
<tr>
<td>6.1.15. Common Table Expressions (“WITH … AS … SELECT”)</td>
<td>286</td>
</tr>
<tr>
<td>6.2. INSERT</td>
<td>290</td>
</tr>
<tr>
<td>6.2.1. INSERT VALUES</td>
<td>291</td>
</tr>
<tr>
<td>6.2.2. INSERT SELECT</td>
<td>292</td>
</tr>
<tr>
<td>6.2.3. INSERT DEFAULT VALUES</td>
<td>293</td>
</tr>
<tr>
<td>6.2.4. The RETURNING clause</td>
<td>293</td>
</tr>
<tr>
<td>6.2.5. Inserting into BLOB columns</td>
<td>294</td>
</tr>
<tr>
<td>6.3. UPDATE</td>
<td>295</td>
</tr>
<tr>
<td>6.3.1. Using an alias</td>
<td>296</td>
</tr>
<tr>
<td>6.3.2. The SET Clause</td>
<td>296</td>
</tr>
<tr>
<td>6.3.3. The WHERE Clause</td>
<td>297</td>
</tr>
<tr>
<td>6.3.4. The ORDER BY and ROWS Clauses</td>
<td>298</td>
</tr>
<tr>
<td>6.3.5. The RETURNING Clause</td>
<td>298</td>
</tr>
<tr>
<td>6.3.6. Updating BLOB columns</td>
<td>299</td>
</tr>
<tr>
<td>6.4. UPDATE OR INSERT</td>
<td>299</td>
</tr>
<tr>
<td>6.4.1. The RETURNING clause</td>
<td>301</td>
</tr>
<tr>
<td>6.4.2. Example of UPDATE OR INSERT</td>
<td>301</td>
</tr>
<tr>
<td>6.5. DELETE</td>
<td>301</td>
</tr>
<tr>
<td>6.5.1. Aliases</td>
<td>302</td>
</tr>
<tr>
<td>6.5.2. WHERE</td>
<td>303</td>
</tr>
<tr>
<td>6.5.3. PLAN</td>
<td>303</td>
</tr>
<tr>
<td>6.5.4. ORDER BY and ROWS</td>
<td>303</td>
</tr>
<tr>
<td>6.5.5. RETURNING</td>
<td>305</td>
</tr>
<tr>
<td>6.6. MERGE</td>
<td>305</td>
</tr>
<tr>
<td>6.6.1. The RETURNING Clause</td>
<td>308</td>
</tr>
<tr>
<td>6.6.2. Examples of MERGE</td>
<td>309</td>
</tr>
<tr>
<td>6.7. EXECUTE PROCEDURE</td>
<td>310</td>
</tr>
<tr>
<td>6.7.1. “Executable” Stored Procedure</td>
<td>311</td>
</tr>
<tr>
<td>6.7.2. Examples of EXECUTE PROCEDURE</td>
<td>311</td>
</tr>
<tr>
<td>6.8. EXECUTE BLOCK</td>
<td>312</td>
</tr>
</tbody>
</table>
7. Procedural SQL (PSQL) Statements ................................................. 315
  7.1. Elements of PSQL ................................................................. 315
    7.1.1. DML Statements with Parameters .................................. 315
    7.1.2. Transactions ................................................................. 315
    7.1.3. Module Structure ............................................................ 315
  7.2. Stored Procedures ............................................................... 318
    7.2.1. Benefits of Stored Procedures ....................................... 319
    7.2.2. Types of Stored Procedures .......................................... 319
    7.2.3. Creating a Stored Procedure ......................................... 320
    7.2.4. Modifying a Stored Procedure ....................................... 320
    7.2.5. Deleting a Stored Procedure ....................................... 320
  7.3. Stored Functions ............................................................... 320
    7.3.1. Creating a Stored Function .......................................... 320
    7.3.2. Modifying a Stored Function ....................................... 320
    7.3.3. Deleting a Stored Function .......................................... 320
  7.4. PSQL Blocks ................................................................. 320
  7.5. Packages ................................................................. 321
    7.5.1. Benefits of Packages .................................................. 321
    7.5.2. Creating a Package .................................................... 322
    7.5.3. Modifying a Package .................................................. 323
    7.5.4. Deleting a Package .................................................... 323
  7.6. Triggers ................................................................. 323
    7.6.1. Firing Order (Order of Execution) ................................ 323
    7.6.2. DML Triggers ............................................................... 323
    7.6.3. Database Triggers ....................................................... 324
    7.6.4. DDL Triggers ............................................................... 325
    7.6.5. Creating Triggers ....................................................... 326
    7.6.6. Modifying Triggers ..................................................... 326
    7.6.7. Deleting a Trigger ..................................................... 326
  7.7. Writing the Body Code ....................................................... 326
    7.7.1. Assignment Statements ............................................... 326
    7.7.2. DECLARE VARIABLE .................................................... 327
    7.7.3. DECLARE .. CURSOR ..................................................... 329
    7.7.4. DECLARE FUNCTION ..................................................... 333
    7.7.5. DECLARE PROCEDURE .................................................. 335
    7.7.6. BEGIN ⋯ END ............................................................... 337
    7.7.7. IF ⋯ THEN ⋯ ELSE ....................................................... 338
    7.7.8. WHILE ⋯ DO ............................................................... 340
# Table of Contents

8. Built-in Scalar Functions

## 8.1. Context Functions

8.1.1. RDB$GET_CONTEXT()

8.1.2. RDB$SET_CONTEXT()

8.2. Mathematical Functions

8.2.1. ABS()

8.2.2. ACOS()

8.2.3. ACOSH()

8.2.4. ASIN()

8.2.5. ASINH()

8.2.6. ATAN()

8.2.7. ATAN2()

8.2.8. ATANH()

8.2.9. CEIL(), CEILING()

8.2.10. COS()

8.2.11. COSH()

8.2.12. COT()

8.2.13. EXP()

8.2.14. FLOOR()

8.2.15. LN()

8.2.16. LOG()

8.2.17. LOG10()
8.2.18. MOD() .................................................. 392
8.2.19. PI() .................................................. 392
8.2.20. POWER() ............................................. 393
8.2.21. RAND() .............................................. 393
8.2.22. ROUND() ............................................ 393
8.2.23. SIGN() ............................................... 395
8.2.24. SIN() ............................................... 395
8.2.25. SINH() .............................................. 396
8.2.26. SQRT() ............................................. 396
8.2.27. TAN() ............................................... 397
8.2.28. TANH() ............................................. 397
8.2.29. TRUNC() .......................................... 398

8.3. String Functions .................................. 399
8.3.1. ASCII_CHAR() .................................... 399
8.3.2. ASCII_VAL() ..................................... 399
8.3.3. BIT_LENGTH() .................................... 400
8.3.4. CHAR_LENGTH(), CHARACTER_LENGTH() 401
8.3.5. HASH() ............................................. 402
8.3.6. LEFT() ............................................. 402
8.3.7. LOWER() .......................................... 403
8.3.8. LPAD() ............................................ 404
8.3.9. OCTET_LENGTH() ................................ 405
8.3.10. OVERLAY() ...................................... 406
8.3.11. POSITION() ..................................... 408
8.3.12. REPLACE() ....................................... 409
8.3.13. REVERSE() ...................................... 410
8.3.14. RIGHT() ......................................... 410
8.3.15. RPAD() .......................................... 411
8.3.16. SUBSTRING() ................................... 412
8.3.17. TRIM() .......................................... 414
8.3.18. UPPER() ....................................... 416

8.4. Date and Time Functions ......................... 416
8.4.1. DATEADD() ..................................... 416
8.4.2. DATEDIFF() ..................................... 418
8.4.3. EXTRACT() ....................................... 419

8.5. Type Casting Functions ......................... 421
8.5.1. CAST() ........................................... 421

8.6. Bitwise Functions ................................ 424
8.6.1. BIN_AND() ..................................... 424
8.6.2. BIN_NOT() ...................................... 425
8.6.3. BIN_OR() ....................................... 426
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10. Window (Analytical) Functions</td>
<td>456</td>
</tr>
<tr>
<td>10.1. Aggregate Functions as Window Functions</td>
<td>457</td>
</tr>
<tr>
<td>10.2. Partitioning</td>
<td>457</td>
</tr>
<tr>
<td>10.3. Ordering</td>
<td>458</td>
</tr>
<tr>
<td>10.4. Ranking Functions</td>
<td>459</td>
</tr>
<tr>
<td>10.4.1. DENSE_RANK</td>
<td>460</td>
</tr>
<tr>
<td>10.4.2. RANK</td>
<td>461</td>
</tr>
<tr>
<td>10.4.3. ROW_NUMBER</td>
<td>461</td>
</tr>
<tr>
<td>10.5. Navigational Functions</td>
<td>462</td>
</tr>
<tr>
<td>10.5.1. FIRST_VALUE</td>
<td>463</td>
</tr>
<tr>
<td>10.5.2. LAG</td>
<td>464</td>
</tr>
<tr>
<td>10.5.3. LAST_VALUE</td>
<td>465</td>
</tr>
<tr>
<td>10.5.4. LEAD</td>
<td>465</td>
</tr>
<tr>
<td>10.5.5. NTH_VALUE</td>
<td>466</td>
</tr>
<tr>
<td>10.6. Aggregate Functions Inside Window Specification</td>
<td>467</td>
</tr>
<tr>
<td>11. Context Variables</td>
<td>468</td>
</tr>
<tr>
<td>11.1. CURRENT_CONNECTION</td>
<td>468</td>
</tr>
<tr>
<td>11.2. CURRENT_DATE</td>
<td>468</td>
</tr>
<tr>
<td>11.3. CURRENT_ROLE</td>
<td>469</td>
</tr>
<tr>
<td>11.4. CURRENT_TIME</td>
<td>469</td>
</tr>
<tr>
<td>11.5. CURRENT_TIMESTAMP</td>
<td>470</td>
</tr>
<tr>
<td>11.6. CURRENT TRANSACTION</td>
<td>472</td>
</tr>
<tr>
<td>11.7. CURRENT_USER</td>
<td>472</td>
</tr>
<tr>
<td>11.8. DELETING</td>
<td>473</td>
</tr>
<tr>
<td>11.9. GDSCODE</td>
<td>473</td>
</tr>
<tr>
<td>11.10. INSERTING</td>
<td>474</td>
</tr>
<tr>
<td>11.11. LOCALTIME</td>
<td>474</td>
</tr>
<tr>
<td>11.12. LOCALTIMESTAMP</td>
<td>475</td>
</tr>
<tr>
<td>11.13. NEW</td>
<td>476</td>
</tr>
<tr>
<td>11.14. 'NOW'</td>
<td>477</td>
</tr>
<tr>
<td>11.15. OLD</td>
<td>478</td>
</tr>
<tr>
<td>11.16. ROW_COUNT</td>
<td>479</td>
</tr>
<tr>
<td>11.17. SQLCODE</td>
<td>479</td>
</tr>
<tr>
<td>11.18. SQLSTATE</td>
<td>480</td>
</tr>
<tr>
<td>11.19. 'TODAY'</td>
<td>481</td>
</tr>
<tr>
<td>11.20. 'TOMORROW'</td>
<td>482</td>
</tr>
<tr>
<td>11.21. UPDATING</td>
<td>482</td>
</tr>
<tr>
<td>11.22. 'YESTERDAY'</td>
<td>483</td>
</tr>
<tr>
<td>11.23. USER</td>
<td>483</td>
</tr>
<tr>
<td>12. Transaction Control</td>
<td>485</td>
</tr>
</tbody>
</table>
# 12. Transaction Statements
- 12.1.1. `SET TRANSACTION` .................................................. 485
- 12.1.2. `COMMIT` .......................................................... 492
- 12.1.3. `ROLLBACK` ......................................................... 494
- 12.1.4. `SAVEPOINT` ..................................................... 495
- 12.1.5. `RELEASE SAVEPOINT` ......................................... 496
- 12.1.6. Internal Savepoints .............................................. 497
- 12.1.7. Savepoints and PSQL ............................................ 497

# 13. Security
- 13.1. User Authentication .................................................. 499
  - 13.1.1. Specially Privileged Users ..................................... 500
  - 13.1.2. `RDB$ADMIN` Role ............................................. 501
  - 13.1.3. Administrators ................................................. 505
- 13.2. SQL Statements for User Management .......................... 506
  - 13.2.1. `CREATE USER` .................................................. 507
  - 13.2.2. `ALTER USER` ................................................. 510
  - 13.2.3. `CREATE OR ALTER USER` .................................... 512
  - 13.2.4. `DROP USER` .................................................... 513
- 13.3. SQL Privileges ........................................................ 514
  - 13.3.1. The Object Owner ............................................... 514
- 13.4. `ROLE` .................................................................. 515
  - 13.4.1. `CREATE ROLE` .................................................. 515
  - 13.4.2. `ALTER ROLE` .................................................. 516
  - 13.4.3. `DROP ROLE` .................................................... 517
- 13.5. Statements for Granting Privileges ............................... 517
  - 13.5.1. `GRANT` .......................................................... 517
- 13.6. Statements for Revoking Privileges ............................... 527
  - 13.6.1. `REVOKE` .......................................................... 527
- 13.7. Mapping of Users to Objects ....................................... 532
  - 13.7.1. The Mapping Rule ............................................... 532
  - 13.7.2. `CREATE MAPPING` .............................................. 533
  - 13.7.3. `ALTER MAPPING` .............................................. 536
  - 13.7.4. `CREATE OR ALTER MAPPING` .............................. 537
  - 13.7.5. `DROP MAPPING` ............................................... 537
- 13.8. Database Encryption .................................................. 538
  - 13.8.1. Encrypting a Database ......................................... 539
  - 13.8.2. Decrypting a Database ....................................... 539

# 14. Management Statements
- 14.1. Changing the Current Role ......................................... 541
  - 14.1.1. `SET ROLE` ....................................................... 541
  - 14.1.2. `SET TRUSTED ROLE` ......................................... 542
Chapter 1. About the Firebird 3.0 Language Reference

This Language Reference describes the SQL language supported by Firebird 3.0.

This Firebird 3.0 Language Reference is the second comprehensive manual to cover all aspects of the query language used by developers to communicate, through their applications, with the Firebird relational database management system.

1.1. Subject

The subject of this volume is wholly Firebird’s implementation of the SQL relational database language. Firebird conforms closely with international standards for SQL, from data type support, data storage structures, referential integrity mechanisms, to data manipulation capabilities and access privileges. Firebird also implements a robust procedural language—procedural SQL (PSQL)—for stored procedures, triggers and dynamically-executable code blocks. These are the areas addressed in this volume.

This document does not cover configuration of Firebird, Firebird command-line tools, nor its programming APIs.

1.2. Authorship

For the Firebird 3.0 version, the Firebird 2.5 Language Reference was taken as the base, and Firebird 3.0 information was added based on the Firebird 3.0 release notes, feature documentation, and the Russian Firebird 3.0 Language Reference. This document, however, is not a direct translation of the Russian Firebird 3.0 Language Reference.

1.2.1. Contributors

Direct Content

- Dmitry Filippov (writer)
- Alexander Karpeykin (writer)
- Alexey Kovyazin (writer, editor)
- Dmitry Kuzmenko (writer, editor)
- Denis Simonov (writer, editor)
- Paul Vinkenoog (writer, designer)
- Dmitry Yemanov (writer)
- Mark Rotteveel (writer)

Resource Content

- Adriano dos Santos Fernandes
1.3. Acknowledgments

Sponsors and Other Donors

See also the Firebird 2.5 Language Reference for its donors.


Moscow Exchange (Russia)

*Moscow Exchange is the largest exchange holding in Russia and Eastern Europe, founded on December 19, 2011, through the consolidation of the MICEX (founded in 1992) and RTS (founded in 1995) exchange groups. Moscow Exchange ranks among the world’s top 20 exchanges by trading in bonds and by the total capitalization of shares traded, as well as among the 10 largest exchange platforms for trading derivatives.*

IBSurgeon (ibase.ru) (Russia)

*Technical support and developer of administrator tools for the Firebird DBMS.*

1.4. Contributing

There are several ways you can contribute to the documentation of Firebird, or Firebird in general:

- Participate on the mailing lists (see [https://www.firebirdsql.org/en/mailing-lists/](https://www.firebirdsql.org/en/mailing-lists/))
- Report bugs or submit pull requests on GitHub ([https://github.com/FirebirdSQL/](https://github.com/FirebirdSQL/))
- Become a developer (contact us on [firebird-devel](mailto:firebird-devel))
- Donate to the Firebird Foundation (see [https://www.firebirdsql.org/en/donate/](https://www.firebirdsql.org/en/donate/))
- Become a paying member or sponsor of the Firebird Foundation (see [https://www.firebirdsql.org/en/firebird-foundation/](https://www.firebirdsql.org/en/firebird-foundation/))
Chapter 2. SQL Language Structure

This reference describes the SQL language supported by Firebird.

2.1. Background to Firebird’s SQL Language

To begin, a few points about some characteristics that are in the background to Firebird’s language implementation.

2.1.1. SQL Flavours

Distinct subsets of SQL apply to different sectors of activity. The SQL subsets in Firebird’s language implementation are:

- Dynamic SQL (DSQL)
- Procedural SQL (PSQL)
- Embedded SQL (ESQL)
- Interactive SQL (ISQL)

Dynamic SQL is the major part of the language which corresponds to the Part 2 (SQL/Foundation) part of the SQL specification. DSQL represents statements passed by client applications through the public Firebird API and processed by the database engine.

Procedural SQL augments Dynamic SQL to allow compound statements containing local variables, assignments, conditions, loops and other procedural constructs. PSQL corresponds to the Part 4 (SQL/PSM) part of the SQL specifications. Originally, PSQL extensions were available in persistent stored modules (procedures and triggers) only, but in more recent releases they were surfaced in Dynamic SQL as well (see EXECUTE BLOCK).

Embedded SQL defines the DSQL subset supported by Firebird gpre, the application which allows you to embed SQL constructs into your host programming language (C, C++, Pascal, Cobol, etc.) and preprocess those embedded constructs into the proper Firebird API calls.

Only a portion of the statements and expressions implemented in DSQL are supported in ESQL.

Interactive ISQL refers to the language that can be executed using Firebird isql, the command-line application for accessing databases interactively. As a regular client application, its native language is DSQL. It also offers a few additional commands that are not available outside its specific environment.

Both DSQL and PSQL subsets are completely presented in this reference. Neither ESQL nor ISQL flavours are described here unless mentioned explicitly.

2.1.2. SQL Dialects

SQL dialect is a term that defines the specific features of the SQL language that are available when
accessing a database. SQL dialects can be defined at the database level and specified at the connection level. Three dialects are available:

- **Dialect 1** is intended solely to allow backward compatibility with legacy databases from very old InterBase versions, v.5 and below. Dialect 1 databases retain certain language features that differ from Dialect 3, the default for Firebird databases.
  - Date and time information are stored in a `DATE` data type. A `TIMESTAMP` data type is also available, that is identical to this `DATE` implementation.
  - Double quotes may be used as an alternative to apostrophes for delimiting string data. This is contrary to the SQL standard—double quotes are reserved for a distinct syntactic purpose both in standard SQL and in Dialect 3. Double-quoting strings is therefore to be avoided strenuously.
  - The precision for `NUMERIC` and `DECIMAL` data types is smaller than in Dialect 3 and, if the precision of a fixed decimal number is greater than 9, Firebird stores it internally as a long floating point value.
  - The `BIGINT` (64-bit integer) data type is not supported.
  - Identifiers are case-insensitive and must always comply with the rules for regular identifiers—see the section Identifiers below.
  - Although generator values are stored as 64-bit integers, a Dialect 1 client request, `SELECT GEN_ID (MyGen, 1)` for example, will return the generator value truncated to 32 bits.

- **Dialect 2** is available only on the Firebird client connection and cannot be set in the database. It is intended to assist debugging of possible problems with legacy data when migrating a database from dialect 1 to 3.

- **Dialect 3** databases,
  - numbers (`DECIMAL` and `NUMERIC` data types) are stored internally as long fixed point values (scaled integers) when the precision is greater than 9.
  - The `TIME` data type is available for storing time-of-day data only.
  - The `DATE` data type stores only date information.
  - The 64-bit integer data type `BIGINT` is available.
  - Double quotes are reserved for delimiting non-regular identifiers, enabling object names that are case-sensitive or that do not meet the requirements for regular identifiers in other ways.
  - All strings must be delimited with single quotes (apostrophes).
  - Generator values are stored as 64-bit integers.

⚠️ Use of Dialect 3 is strongly recommended for newly developed databases and applications. Both database and connection dialects should match, except under migration conditions with Dialect 2.

This reference describes the semantics of SQL Dialect 3 unless specified otherwise.
2.1.3. Error Conditions

Processing of every SQL statement either completes successfully or fails due to a specific error condition. Error handling can be done as in the client the application and on the server side using PSQL.

2.2. Basic Elements: Statements, Clauses, Keywords

The primary construct in SQL is the statement. A statement defines what the database management system should do with a particular data or metadata object. More complex statements contain simpler constructs — clauses and options.

Clauses

A clause defines a certain type of directive in a statement. For instance, the WHERE clause in a SELECT statement and in some other data manipulation statements (UPDATE, DELETE) specifies criteria for searching one or more tables for the rows that are to be selected, updated or deleted. The ORDER BY clause specifies how the output data — result set — should be sorted.

Options

Options, being the simplest constructs, are specified in association with specific keywords to provide qualification for clause elements. Where alternative options are available, it is usual for one of them to be the default, used if nothing is specified for that option. For instance, the SELECT statement will return all of the rows that match the search criteria unless the DISTINCT option restricts the output to non-duplicated rows.

Keywords

All words that are included in the SQL lexicon are keywords. Some keywords are reserved, meaning their usage as identifiers for database objects, parameter names or variables is prohibited in some or all contexts. Non-reserved keywords can be used as identifiers, although it is not recommended. From time to time, non-reserved keywords may become reserved when some new language feature is introduced.

For instance, the following statement will be executed without errors because, although ABS is a keyword, it is not a reserved word.

```
CREATE TABLE T (ABS INT NOT NULL);
```

On the contrary, the following statement will return an error because ADD is both a keyword and a reserved word.

```
CREATE TABLE T (ADD INT NOT NULL);
```

Refer to the list of reserved words and keywords in the chapter Reserved Words and Keywords.
2.3. Identifiers

All database objects have names, often called *identifiers*. The maximum identifier length is 31 bytes. Two types of names are valid as identifiers: *regular* names, similar to variable names in regular programming languages, and *delimited* names that are specific to SQL. To be valid, each type of identifier must conform to a set of rules, as follows:

### 2.3.1. Rules for Regular Object Identifiers

- Length cannot exceed 31 characters
- The name must start with an unaccented, 7-bit ASCII alphabetic character. It may be followed by other 7-bit ASCII letters, digits, underscores or dollar signs. No other characters, including spaces, are valid. The name is case-insensitive, meaning it can be declared and used in either upper or lower case. Thus, from the system’s point of view, the following names are the same:

  ```
  fullname
  FULLNAME
  FullNaMe
  FullName
  ```

*Regular name syntax*

```text
<name> ::= <letter> | <name><letter> | <name><digit> | <name>_ | <name>$
<letter> ::= <upper letter> | <lower letter>
<upper letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
                N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<lower letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
                 n | o | p | q | r | s | t | u | v | w | x | y | z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

### 2.3.2. Rules for Delimited Object Identifiers

- Length cannot exceed 31 bytes. Identifiers are stored in character set `UNICODE_FSS`, which means characters outside the ASCII range are stored using 2 or 3 bytes.
- The entire string must be enclosed in double-quotes, e.g. "anIdentifier"
- It may contain any character from the `UNICODE_FSS` character set, including accented characters, spaces and special characters
- An identifier can be a reserved word
- Delimited identifiers are case-sensitive in all contexts
- Trailing spaces in delimited names are removed, as with any string constant
Delimited identifiers are available in Dialect 3 only. For more details on dialects, see SQL Dialects.

**Delimited name syntax**

<delimited name> ::= "<permitted_character>[<permitted_character> ...]"

A delimited identifier such as "FULLNAME" is the same as the regular identifiers FULLNAME, fullname, FullName, and so on. The reason is that Firebird stores regular identifiers in upper case, regardless of how they were defined or declared. Delimited identifiers are always stored according to the exact case of their definition or declaration. Thus, "FullName" (quoted) is different from FullName (unquoted, i.e. regular) which is stored as FULLNAME in the metadata.

### 2.4. Literals

Literals are used to directly represent data. Examples of standard types of literals are:

- **integer** - 0, -34, 45, 0X080000000;
- **fixed-point** - 0.0, -3.14
- **floating-point** - 3.23e-23;
- **string** - 'text', 'don''t!';
- **binary string** - x'48656C6C6F20776F726C64'
- **date** - DATE '2018-01-19';
- **time** - TIME '15:12:56';
- **timestamp** - TIMESTAMP '2018-01-19 13:32:02';
- **boolean** - true, false, unknown
- **null state** - null

Details about handling the literals for each data type are discussed in the next chapter, Data Types and Subtypes.

### 2.5. Operators and Special Characters

A set of special characters is reserved for use as operators or separators.

<special char> ::=<space> | " | | % | & | ' | ( | ) | * | + | , | - |
| . | / | : | ; | | < | = | > | ? | [ | ] | ^ | { | }

Some of these characters, alone or in combinations, may be used as operators (arithmetical, string, logical), as SQL command separators, to quote identifiers, and to mark the limits of string literals or comments.
### Operator Syntax

\[
<\text{operator}> ::= \\
\text{<string concatenation operator>}
\ |
\text{<arithmetic operator>}
\ |
\text{<comparison operator>}
\ |
\text{<logical operator>}
\]

\[
<\text{string concatenation operator}> ::= "||"
\]

\[
<\text{arithmetic operator}> ::= * | / | + | - |
\]

\[
<\text{comparison operator}> ::= \\
= | <> | != | ^= | >= | <= \\
| !> | ~> | ^> | !< | ~< | ^<
\]

\[
<\text{logical operator}> ::= \text{NOT} | \text{AND} | \text{OR}
\]

For more details on operators, see [Expressions](#).

### 2.6. Comments

Comments may be present in SQL scripts, SQL statements and PSQL modules. A comment can be any text specified by the code writer, usually used to document how particular parts of the code work. The parser ignores the text of comments.

Firebird supports two types of comments: **block** and **in-line**.

**Syntax**

\[
<\text{comment}> ::= <\text{block comment}> | <\text{single-line comment}>
\]

\[
<\text{block comment}> ::= \\
/* <\text{character}>[<\text{character}> ...] */
\]

\[
<\text{single-line comment}> ::= \\
-- <\text{character}>[<\text{character}> ...]<\text{end line}>
\]

Block comments start with the /* character pair and end with the */ character pair. Text in block comments may be of any length and can occupy multiple lines.

In-line comments start with a pair of hyphen characters, -- and continue up to the end of the current line.

**Example**

```sql
CREATE PROCEDURE P(APARAM INT)
    RETURNS (B INT)
AS
```
BEGIN
/* This text will be ignored during the execution of the statement since it is a comment */
B = A + 1; -- In-line comment
SUSPEND;
END
Chapter 3. Data Types and Subtypes

Data of various types are used to:

- define columns in a database table in the `CREATE TABLE` statement or change columns using `ALTER TABLE`
- declare or change a `domain` using the `CREATE DOMAIN` or `ALTER DOMAIN` statements
- declare local variables in stored procedures, PSQL blocks and triggers and specify parameters in stored procedures
- indirectly specify arguments and return values when declaring external functions (UDFs — user-defined functions)
- provide arguments for the `CAST()` function when explicitly converting data from one type to another

Table 1. Overview of Data Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Precision &amp; Limits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>64 bits</td>
<td>From $-2^{63}$ to $(2^{63} - 1)$</td>
<td>The data type is available in Dialect 3 only</td>
</tr>
<tr>
<td>BLOB</td>
<td>Varying</td>
<td>The size of a BLOB segment is limited to 64K. The maximum size of a BLOB field is 4 GB</td>
<td>A data type of variable size for storing large amounts of data, such as images, text, digital sounds. The basic structural unit is a segment. The blob subtype defines its content</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>8 bits</td>
<td>false, true, unknown</td>
<td>Boolean data type</td>
</tr>
<tr>
<td>CHAR($n$), CHARACTER($n$)</td>
<td>$n$ characters. Size in bytes depends on the encoding, the number of bytes in a character from 1 to 32,767 bytes</td>
<td>A fixed-length character data type. When its data is displayed, trailing spaces are added to the string up to the specified length. Trailing spaces are not stored in the database but are restored to match the defined length when the column is displayed on the client side. Network traffic is reduced by not sending spaces over the LAN. If the number of characters is not specified, 1 is used by default.</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>32 bits</td>
<td>From 0001-01-01 AD to 9999-12-31 AD</td>
<td>ISC_DATE. Date only, no time element</td>
</tr>
<tr>
<td>Name</td>
<td>Size</td>
<td>Precision &amp; Limits</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DECIMAL (precision, scale)</td>
<td>Varying (16, 32 or 64 bits)</td>
<td><strong>precision</strong> = from 1 to 18, defines the least possible number of digits to store; <strong>scale</strong> = from 0 to 18, defines the number of digits after the decimal point</td>
<td>A number with a decimal point that has <strong>scale</strong> digits after the point. <strong>scale</strong> must be less than or equal to <strong>precision</strong>. Example: DECIMAL(10,3) contains a number in exactly the following format: ppppppp.sss</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>64 bits</td>
<td>2.225 * 10^{308} to 1.797 * 10^{308}</td>
<td>Double-precision IEEE, ~15 digits, reliable size depends on the platform</td>
</tr>
<tr>
<td>FLOAT</td>
<td>32 bits</td>
<td>1.175 * 10^{-38} to 3.402 * 10^{38}</td>
<td>Single-precision IEEE, ~7 digits</td>
</tr>
<tr>
<td>INTEGER, INT</td>
<td>32 bits</td>
<td>-2,147,483,648 up to 2,147,483,647</td>
<td>Signed long</td>
</tr>
<tr>
<td>NUMERIC (precision, scale)</td>
<td>Varying (16, 32 or 64 bits)</td>
<td><strong>precision</strong> = from 1 to 18, defines the exact number of digits to store; <strong>scale</strong> = from 0 to 18, defines the number of digits after the decimal point</td>
<td>A number with a decimal point that has <strong>scale</strong> digits after the point. <strong>scale</strong> must be less than or equal to <strong>precision</strong>. Example: NUMERIC(10,3) contains a number in exactly the following format: ppppppp.sss</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>16 bits</td>
<td>-32,768 to 32,767</td>
<td>Signed short (word)</td>
</tr>
<tr>
<td>TIME</td>
<td>32 bits</td>
<td>0:00 to 23:59:59.9999</td>
<td>ISC_TIME. Time of day. It cannot be used to store an interval of time</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>64 bits (2 X 32 bits)</td>
<td>From start of day 0001-01-01 AD to end of day 9999-12-31 AD</td>
<td>Date and time of day</td>
</tr>
<tr>
<td>Name</td>
<td>Size</td>
<td>Precision &amp; Limits</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>---------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>VARCHAR(n), CHAR VARYING, CHARACTER VARYING</td>
<td>n characters. Size in bytes depends on the encoding, the number of bytes in a character</td>
<td>from 1 to 32,765 bytes</td>
<td>Variable length string type. The total size of characters in bytes cannot be larger than (32KB-3), taking into account their encoding. The two trailing bytes store the declared length. There is no default size: the n argument is mandatory. Leading and trailing spaces are stored and they are not trimmed, except for those trailing characters that are past the declared length.</td>
</tr>
</tbody>
</table>

### Note About Dates

Bear in mind that a time series consisting of dates in past centuries is processed without taking into account the actual historical facts, as though the Gregorian calendar were applicable throughout the entire series.

### 3.1. Integer Data Types

The SMALLINT, INTEGER and BIGINT data types are used for integers of various precision in Dialect 3. Firebird does not support an unsigned integer data type.

#### 3.1.1. SMALLINT

The 16-bit SMALLINT data type is for compact data storage of integer data for which only a narrow range of possible values is required. Numbers of the SMALLINT type are within the range from \(-2^{16}\) to \(2^{16} - 1\), that is, from -32,768 to 32,767.

**SMALLINT Examples**

```sql
CREATE DOMAIN DFLAG AS SMALLINT DEFAULT 0 NOT NULL CHECK (VALUE=-1 OR VALUE=0 OR VALUE=1);
CREATE DOMAIN RGB_VALUE AS SMALLINT;
```

#### 3.1.2. INTEGER

The INTEGER data type is a 32-bit integer. The shorthand name of the data type is INT. Numbers of the INTEGER type are within the range from \(-2^{32}\) to \(2^{32} - 1\), that is, from -2,147,483,648 to 2,147,483,647.

**INTEGER Example**

```sql
CREATE TABLE CUSTOMER (
    CUST_NO INTEGER NOT NULL,
    CUSTOMER VARCHAR(25) NOT NULL,
);```
3.1.3. BIGINT

BIGINT is an SQL:99-compliant 64-bit integer data type, available only in Dialect 3. If a client uses Dialect 1, the generator value sent by the server is reduced to a 32-bit integer (INTEGER). When Dialect 3 is used for connection, the generator value is of type BIGINT.

Numbers of the BIGINT type are within the range from \(-2^{63}\) to \(2^{63} - 1\), or from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

3.1.4. Hexadecimal Format for Integer Numbers

Starting from Firebird 2.5, constants of the three integer types can be specified in hexadecimal format by means of 9 to 16 hexadecimal digits for BIGINT or 1 to 8 digits for INTEGER. Hex representation for writing to SMALLINT is not explicitly supported but Firebird will transparently convert a hex number to SMALLINT if necessary, provided it falls within the ranges of negative and positive SMALLINT.

The usage and numerical value ranges of hexadecimal notation are described in more detail in the discussion of number constants in the chapter entitled Common Language Elements.

Examples Using Integer Types

```sql
CREATE TABLE WHOLELOTTARECORDS (  
    ID BIGINT NOT NULL PRIMARY KEY,  
    DESCRIPTION VARCHAR(32)  
);  

INSERT INTO MYBIGINTS VALUES (  
    -236453287458723,  
    328832607832,  
    22,  
    -56786237632476,  
    0X6F55A09D42, -- 478177959234  
    0X7FFFFFFFFFFFFFFF, -- 9223372036854775807  
    0XFFFFFFFFFFFFFFFF, -- -1  
    0X80000000, -- -2147483648, an INTEGER  
    0X80000000, -- 2147483648, a BIGINT  
    0XFFFFFFFF, -- -1, an INTEGER  
    0XFFFFFFFF -- 4294967295, a BIGINT  
);  
```

The hexadecimal INTEGERs in the above example are automatically cast to BIGINT before being inserted into the table. However, this happens after the numerical value is determined, so 0x80000000 (8 digits) and 0x0800000000 (9 digits) will be saved as different BIGINT values.
3.2. Floating-Point Data Types

Floating point data types are stored in an IEEE 754 binary format that comprises sign, exponent and mantissa. Precision is dynamic, corresponding to the physical storage format of the value, which is exactly 4 bytes for the FLOAT type and 8 bytes for DOUBLE PRECISION.

Considering the peculiarities of storing floating-point numbers in a database, these data types are not recommended for storing monetary data. For the same reasons, columns with floating-point data are not recommended for use as keys or to have uniqueness constraints applied to them.

For testing data in columns with floating-point data types, expressions should check using a range, for instance, BETWEEN, rather than searching for exact matches.

When using these data types in expressions, extreme care is advised regarding the rounding of evaluation results.

3.2.1. FLOAT

The FLOAT data type has an approximate precision of 7 digits after the decimal point. To ensure the safety of storage, rely on 6 digits.

3.2.2. DOUBLE PRECISION

The DOUBLE PRECISION data type is stored with an approximate precision of 15 digits.

3.3. Fixed-Point Data Types

Fixed-point data types ensure the predictability of multiplication and division operations, making them the choice for storing monetary values. Firebird implements two fixed-point data types: NUMERIC and DECIMAL. According to the standard, both types limit the stored number to the declared scale (the number of digits after the decimal point).

Different treatments limit precision for each type: precision for NUMERIC columns is exactly “as declared”, while DECIMAL columns accepts numbers whose precision is at least equal to what was declared.

The behaviour of NUMERIC and DECIMAL in Firebird is like the SQL-standard DECIMAL; the precision is at least equal to what was declared.

For instance, NUMERIC(4, 2) defines a number consisting altogether of four digits, including two digits after the decimal point; that is, it can have up to two digits before the point and no more than two digits after the point. If the number 3.1415 is written to a column with this data type definition, the value of 3.14 will be saved in the NUMERIC(4, 2) column.

The form of declaration for fixed-point data, for instance, NUMERIC(p, s), is common to both types. It is important to realise that the s argument in this template is scale, rather than “a count of digits after the decimal point”. Understanding the mechanism for storing and retrieving fixed-point data should help to visualise why: for storage, the number is multiplied by 10^s (10 to the power of s).
converting it to an integer; when read, the integer is converted back.

The method of storing fixed-point data in the DBMS depends on several factors: declared precision, database dialect, declaration type.

### Table 2. Method of Physical Storage for Real Numbers

<table>
<thead>
<tr>
<th>Precision</th>
<th>Data type</th>
<th>Dialect 1</th>
<th>Dialect 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 4</td>
<td>NUMERIC</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>1 - 4</td>
<td>DECIMAL</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>5 - 9</td>
<td>NUMERIC or DECIMAL</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>10 - 18</td>
<td>NUMERIC or DECIMAL</td>
<td>DOUBLE PRECISION</td>
<td>BIGINT</td>
</tr>
</tbody>
</table>

#### 3.3.1. NUMERIC

**Data Declaration Format**

```
NUMERIC
| NUMERIC(precision)
| NUMERIC(precision, scale)
```

### Table 3. NUMERIC Type Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>precision</td>
<td>Precision, between 1 and 18. Defaults to 9.</td>
</tr>
<tr>
<td>scale</td>
<td>Scale, between 0 and scale. Defaults to 0.</td>
</tr>
</tbody>
</table>

**Storage Examples**

Further to the explanation above, the DBMS will store NUMERIC data according the declared `precision` and `scale`. Some more examples are:

- NUMERIC(4) stored as SMALLINT (exact data)
- NUMERIC(4,2) SMALLINT (data * 10^2)
- NUMERIC(10,4) (Dialect 1) DOUBLE PRECISION (Dialect 3) BIGINT (data * 10^4)

Always keep in mind that the storage format depends on the precision. For instance, you define the column type as NUMERIC(2,2) presuming that its range of values will be -0.99...0.99. However, the actual range of values for the column will be -327.68..327.67, which is due to storing the NUMERIC(2,2) data type in the SMALLINT format. In storage, the NUMERIC(4,2), NUMERIC(3,2) and NUMERIC(2,2) data types are the same, in fact. It means that if you really want to store data in a column with the NUMERIC(2,2) data type and limit the range to -0.99...0.99, you will have to create a constraint for it.
3.3.2. DECIMAL

Data Declaration Format

```
DECIMAL
| DECIMAL(precision)
| DECIMAL(precision, scale)
```

Table 4. DECIMAL Type Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>precision</td>
<td>Precision, between 1 and 18. Defaults to 9.</td>
</tr>
<tr>
<td>scale</td>
<td>Scale, between 0 and scale. Defaults to 0.</td>
</tr>
</tbody>
</table>

Storage Examples

The storage format in the database for DECIMAL is very similar to NUMERIC, with some differences that are easier to observe with the help of some more examples:

```
DECIMAL(4) stored as      INTEGER (exact data)
DECIMAL(4,2)              INTEGER (data * 10^2)
DECIMAL(10,4) (Dialect 1) DOUBLE PRECISION
                        (Dialect 3) BIGINT (data * 10^4)
```

3.4. Data Types for Dates and Times

The DATE, TIME and TIMESTAMP data types are used to work with data containing dates and times. Dialect 3 supports all the three types, while Dialect 1 has only DATE. The DATE type in Dialect 3 is “date-only”, whereas the Dialect 1 DATE type stores both date and time-of-day, equivalent to TIMESTAMP in Dialect 3. Dialect 1 has no “date-only” type.

Dialect 1 DATE data can be defined alternatively as TIMESTAMP and this is recommended for new definitions in Dialect 1 databases.

Fractions of Seconds

If fractions of seconds are stored in date and time data types, Firebird stores them to tenths of a second. If a lower granularity is preferred, the fraction can be specified explicitly as thousandths, hundredths or tenths of a second in Dialect 3 databases of ODS 11 or higher.

Some useful knowledge about subseconds precision:

The time-part of a TIME or TIMESTAMP is a 4-byte WORD, with room for decimilliseconds precision and time values are stored as the number of decimillisseconds elapsed since midnight. The actual precision of values stored in or read from time(stamp) functions and variables is:

- CURRENT_TIME defaults to seconds precision and can be specified up to milliseconds precision with CURRENT_TIME (0|1|2|3)
Chapter 3. Data Types and Subtypes

• CURRENT_TIMESTAMP milliseconds precision. Precision from seconds to milliseconds can be specified with CURRENT_TIMESTAMP (0|1|2|3)

• Literal 'NOW': milliseconds precision

• Functions DATEADD() and DATEDIFF() support up to milliseconds precision. Decimal-milliseconds can be specified but they are rounded to the nearest integer before any operation is performed

• The EXTRACT() function returns up to decimal-milliseconds precision with the SECOND and MILLISECOND arguments

• For TIME and TIMESTAMP literals, Firebird happily accepts up to decimal-milliseconds precision, but truncates (not rounds) the time part to the nearest lower or equal millisecond. Try, for example, SELECT TIME '14:37:54.1249' FROM rdb$database

• The ‘+’ and ‘-’ operators work with decimal-millisecond precision, but only within the expression. As soon as something is stored or even just SELECTed from RDB$DATABASE, it reverts to milliseconds precision

Deci-milliseconds precision is rare and is not currently stored in columns or variables. The best assumption to make from all this is that, although Firebird stores TIME and the TIMESTAMP time-part values as the number of deci-milliseconds \(10^{-4}\) seconds elapsed since midnight, the actual precision could vary from seconds to milliseconds.

3.4.1. DATE

The DATE data type in Dialect 3 stores only date without time. The available range for storing data is from January 01, 1 to December 31, 9999.

Dialect 1 has no “date-only” type.

In Dialect 1, date literals without a time part, as well as ‘TODAY’, ‘YESTERDAY’ and ‘TOMORROW’ automatically get a zero time part.

If, for some reason, it is important to you to store a Dialect 1 timestamp literal with an explicit zero time-part, the engine will accept a literal like '2016-12-25 00:00:00.0000'. However, '2016-12-25' would have precisely the same effect, with fewer keystrokes!

3.4.2. TIME

The TIME data type is available in Dialect 3 only. It stores the time of day within the range from 00:00:00.0000 to 23:59:59.9999.

If you need to get the time-part from DATE in Dialect 1, you can use the EXTRACT function.

**Examples Using EXTRACT()**

EXTRACT (HOUR FROM DATE_FIELD)
See also the `EXTRACT()` function in the chapter entitled *Built-in Functions*.

### 3.4.3. TIMESTAMP

The `TIMESTAMP` data type is available in Dialect 3 and Dialect 1. It comprises two 32-bit words — a date-part and a time-part — to form a structure that stores both date and time-of-day. It is the same as the `DATE` type in Dialect 1.

The `EXTRACT` function works equally well with `TIMESTAMP` as with the Dialect 1 `DATE` type.

### 3.4.4. Operations Using Date and Time Values

The method of storing date and time values makes it possible to involve them as operands in some arithmetic operations. In storage, a date value or date-part of a timestamp is represented as the number of days elapsed since “date zero” — November 17, 1898 — whilst a time value or the time-part of a timestamp is represented as the number of seconds (with fractions of seconds taken into account) since midnight.

An example is to subtract an earlier date, time or timestamp from a later one, resulting in an interval of time, in days and fractions of days.

*Table 5. Arithmetic Operations for Date and Time Data Types*

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>Operation</th>
<th>Operand 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>+</td>
<td>TIME</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATE</td>
<td>+</td>
<td>Numeric value $n$</td>
<td>DATE increased by $n$ whole days. Broken values are rounded (not floored) to the nearest integer</td>
</tr>
<tr>
<td>TIME</td>
<td>+</td>
<td>DATE</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TIME</td>
<td>+</td>
<td>Numeric value $n$</td>
<td>TIME increased by $n$ seconds. The fractional part is taken into account</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>+</td>
<td>Numeric value $n$</td>
<td>TIMESTAMP, where the date will advance by the number of days and part of a day represented by number $n$ — so “+2.75” will push the date forward by 2 days and 18 hours</td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>DATE</td>
<td>Number of days elapsed, within the range <code>DECIMAL(9, 0)</code></td>
</tr>
<tr>
<td>DATE</td>
<td>-</td>
<td>Numeric value $n$</td>
<td>DATE reduced by $n$ whole days. Broken values are rounded (not floored) to the nearest integer</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>TIME</td>
<td>Number of seconds elapsed, within the range <code>DECIMAL(9, 4)</code></td>
</tr>
</tbody>
</table>
### Notes

The DATE type is considered as TIMESTAMP in Dialect 1.

*See also*

DATEADD, DATEDIFF

## 3.5. Character Data Types

For working with character data, Firebird has the fixed-length CHAR and the variable-length VARCHAR data types. The maximum size of text data stored in these data types is 32,767 bytes for CHAR and 32,765 bytes for VARCHAR. The maximum number of characters that will fit within these limits depends on the CHARACTER SET being used for the data under consideration. The collation sequence does not affect this maximum, although it may affect the maximum size of any index that involves the column.

If no character set is explicitly specified when defining a character object, the default character set specified when the database was created will be used. If the database does not have a default character set defined, the field gets the character set NONE.

### 3.5.1. Unicode

Most current development tools support Unicode, implemented in Firebird with the character sets UTF8 and UNICODE_FSS. UTF8 comes with collations for many languages. UNICODE_FSS is more limited and is used mainly by Firebird internally for storing metadata. Keep in mind that one UTF8 character occupies up to 4 bytes, thus limiting the size of CHAR fields to 8,191 characters (32,767/4).

The actual “bytes per character” value depends on the range the character belongs to. Non-accented Latin letters occupy 1 byte, Cyrillic letters from the WIN1251 encoding occupy 2 bytes in UTF8, characters from other encodings may occupy up to 4 bytes.

The UTF8 character set implemented in Firebird supports the latest version of the Unicode standard, thus recommending its use for international databases.
3.5.2. Client Character Set

While working with strings, it is essential to keep the character set of the client connection in mind. If there is a mismatch between the character sets of the stored data and that of the client connection, the output results for string columns are automatically re-encoded, both when data are sent from the client to the server and when they are sent back from the server to the client. For example, if the database was created in the WIN1251 encoding but KOI8R or UTF8 is specified in the client’s connection parameters, the mismatch will be transparent.

3.5.3. Special Character Sets

Character set NONE

The character set NONE is a special character set in Firebird. It can be characterized such that each byte is a part of a string, but the string is stored in the system without any clues about what constitutes any character: character encoding, collation, case, etc. are simply unknown. It is the responsibility of the client application to deal with the data and provide the means to interpret the string of bytes in some way that is meaningful to the application and the human user.

Character set OCTETS

Data in OCTETS encoding are treated as bytes that may not actually be interpreted as characters. OCTETS provides a way to store binary data, which could be the results of some Firebird functions. The database engine has no concept of what it is meant to do with a string of bits in OCTETS, other than just store it and retrieve it. Again, the client side is responsible for validating the data, presenting them in formats that are meaningful to the application and its users and handling any exceptions arising from decoding and encoding them.

3.5.4. Collation Sequence

Each character set has a default collation sequence (COLLATE) that specifies the collation order. Usually, it provides nothing more than ordering based on the numeric code of the characters and a basic mapping of upper- and lower-case characters. If some behaviour is needed for strings that is not provided by the default collation sequence and a suitable alternative collation is supported for that character set, a COLLATE collation clause can be specified in the column definition.

A COLLATE collation clause can be applied in other contexts besides the column definition. For greater-than/less-than comparison operations, it can be added in the WHERE clause of a SELECT statement. If output needs to be sorted in a special alphabetic sequence, or case-insensitively, and the appropriate collation exists, then a COLLATE clause can be included with the ORDER BY clause when rows are being sorted on a character field and with the GROUP BY clause in case of grouping operations.

Case-Insensitive Searching

For a case-insensitive search, the UPPER function could be used to convert both the search argument and the searched strings to upper-case before attempting a match:

…
where upper(name) = upper(:flt_name)

For strings in a character set that has a case-insensitive collation available, you can simply apply the collation, to compare the search argument and the searched strings directly. For example, using the WIN1251 character set, the collation PXW_CYRL is case-insensitive for this purpose:

... WHERE FIRST_NAME COLLATE PXW_CYRL >= :FLT_NAME ...
... ORDER BY NAME COLLATE PXW_CYRL

See also
CONTAINING

UTF8 Collation Sequences

The following table shows the possible collation sequences for the UTF8 character set.

Table 6. Collation Sequences for Character Set UTF8

<table>
<thead>
<tr>
<th>Collation</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS_BASIC</td>
<td>Collation works according to the position of the character in the table (binary). Added in Firebird 2.0</td>
</tr>
<tr>
<td>UNICODE</td>
<td>Collation works according to the UCA algorithm (Unicode Collation Algorithm) (alphabetical). Added in Firebird 2.0</td>
</tr>
<tr>
<td>UTF8</td>
<td>The default, binary collation, identical to UCS_BASIC, which was added for SQL compatibility</td>
</tr>
<tr>
<td>UNICODE_CI</td>
<td>Case-insensitive collation, works without taking character case into account. Added in Firebird 2.1</td>
</tr>
<tr>
<td>UNICODE_CI_AI</td>
<td>Case-insensitive, accent-insensitive collation, works alphabetically without taking character case or accents into account. Added in Firebird 2.5</td>
</tr>
</tbody>
</table>

Example

An example of collation for the UTF8 character set without taking into account the case or accentuation of characters (similar to COLLATE PXW_CYRL).

... ORDER BY NAME COLLATE UNICODE_CI_AI

3.5.5. Character Indexes

In Firebird earlier than version 2.0, a problem can occur with building an index for character columns that use a non-standard collation sequence: the length of an indexed field is limited to 252 bytes with no COLLATE specified or 84 bytes if COLLATE is specified. Multi-byte character sets and
compound indexes limit the size even further.

Starting from Firebird 2.0, the maximum length for an index equals one quarter of the page size, i.e. from 1,024 — for page size 4,096 — to 4,096 bytes — for page size 16,384. The maximum length of an indexed string is 9 bytes less than that quarter-page limit.

**Calculating Maximum Length of an Indexed String Field**

The following formula calculates the maximum length of an indexed string (in characters):

\[
\text{max_char_length} = \text{FLOOR}((\text{page_size} / 4 - 9) / N)
\]

where \(N\) is the number of bytes per character in the character set.

The table below shows the maximum length of an indexed string (in characters), according to page size and character set, calculated using this formula.

**Table 7. Maximum Index Lengths by Page Size and Character Size**

<table>
<thead>
<tr>
<th>Page Size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,096</td>
<td>1,015</td>
<td>507</td>
<td>338</td>
<td>253</td>
<td>169</td>
</tr>
<tr>
<td>8,192</td>
<td>2,039</td>
<td>1,019</td>
<td>679</td>
<td>509</td>
<td>339</td>
</tr>
<tr>
<td>16,384</td>
<td>4,087</td>
<td>2,043</td>
<td>1,362</td>
<td>1,021</td>
<td>682</td>
</tr>
</tbody>
</table>

With case-insensitive collations (“_CI”), one character in the index will occupy not 4, but 6 (six) bytes, so the maximum key length for a page of — for example — 4,096 bytes, will be 169 characters.

See also

**CREATE DATABASE, Collation sequence, SELECT, WHERE, GROUP BY, ORDER BY**

### 3.5.6. Character Types in Detail

**CHAR**

CHAR is a fixed-length data type. If the entered number of characters is less than the declared length, trailing spaces will be added to the field. Generally, the pad character does not have to be a space: it depends on the character set. For example, the pad character for the OCTETS character set is zero.

The full name of this data type is CHARACTER, but there is no requirement to use full names and people rarely do so.

Fixed-length character data can be used to store codes whose length is standard and has a definite “width” in directories. An example of such a code is an EAN13 barcode — 13 characters, all filled.

**Declaration Syntax**

```
{CHAR | CHARACTER} [(length)]
```
If no length is specified, it is taken to be 1.

A valid length is from 1 to the maximum number of characters that can be accommodated within 32,767 bytes.

Formally, the COLLATE clause is not part of the data type declaration, and its position depends on the syntax of the statement.

**VARCHAR**

VARCHAR is the basic string type for storing texts of variable length, up to a maximum of 32,765 bytes. The stored structure is equal to the actual size of the data plus 2 bytes where the length of the data is recorded.

All characters that are sent from the client application to the database are considered meaningful, including the leading and trailing spaces. However, trailing spaces are not stored: they will be restored upon retrieval, up to the recorded length of the string.

The full name of this type is CHARACTER VARYING. Another variant of the name is written as CHAR VARYING.

**Syntax**

```
{VARCHAR | {CHAR | CHARACTER} VARYING} (length)
```

Formally, the COLLATE clause is not part of the data type declaration, and its position depends on the syntax of the statement.

**NCHAR**

NCHAR is a fixed-length character data type with the ISO8859_1 character set predefined. In all other respects it is the same as CHAR.

**Syntax**

```
{NCHAR | NATIONAL {CHAR | CHARACTER}} [(length)]
```

If no length is specified, it is taken to be 1.

A similar data type is available for the variable-length string type: NATIONAL {CHAR | CHARACTER} VARYING.
3.6. Boolean Data Type

Firebird 3.0 introduced a fully-fledged Boolean data type.

3.6.1. BOOLEAN

The SQL:2008 compliant BOOLEAN data type (8 bits) comprises the distinct truth values TRUE and FALSE. Unless prohibited by a NOT NULL constraint, the BOOLEAN data type also supports the truth value UNKNOWN as the null value. The specification does not make a distinction between the NULL value of this data type and the truth value UNKNOWN that is the result of an SQL predicate, search condition, or Boolean value expression: they may be used interchangeably to mean exactly the same thing.

As with many programming languages, the SQL BOOLEAN values can be tested with implicit truth values. For example, field1 OR field2 and NOT field1 are valid expressions.

The IS Operator

Predicates can use the operator Boolean IS [NOT] for matching. For example, field1 IS FALSE, or field1 IS NOT TRUE.

• Equivalence operators (“=”, “!=”, “<>” and so on) are valid in all comparisons.

BOOLEAN Examples

1. Inserting and selecting

```sql
CREATE TABLE TBOOL (ID INT, BVAL BOOLEAN);
COMMIT;

INSERT INTO TBOOL VALUES (1, TRUE);
INSERT INTO TBOOL VALUES (2, 2 = 4);
INSERT INTO TBOOL VALUES (3, NULL = 1);
COMMIT;

SELECT * FROM TBOOL;
ID   BVAL
====== ======
 1   <true>
 2   <false>
 3   <null>
```

2. Test for TRUE value

```sql
SELECT * FROM TBOOL WHERE BVAL;
ID   BVAL
====== ======
 1   <true>
```
3. Test for FALSE value

```sql
SELECT * FROM TBOOL WHERE BVAL IS FALSE;
ID    BVAL
========== ======
2 <false>
```

4. Test for UNKNOWN value

```sql
SELECT * FROM TBOOL WHERE BVAL IS UNKNOWN;
ID    BVAL
========== ======
3 <null>
```

5. Boolean values in SELECT list

```sql
SELECT ID, BVAL, BVAL AND ID < 2
FROM TBOOL;
ID    BVAL
========== ======= ======
1 <true> <true>
2 <false> <false>
3 <null> <false>
```

6. PSQL declaration with start value

```sql
DECLARE VARIABLE VAR1 BOOLEAN = TRUE;
```

7. Valid syntax, but as with a comparison with NULL, will never return any record

```sql
SELECT * FROM TBOOL WHERE BVAL = UNKNOWN;
SELECT * FROM TBOOL WHERE BVAL <> UNKNOWN;
```

**Use of Boolean against other data types**

Although BOOLEAN is not inherently convertible to any other datatype, from version 3.0.1 the strings 'true' and 'false' (case-insensitive) will be implicitly cast to BOOLEAN in value expressions, e.g.

```sql
if (true > 'false') then ...
```

'false' is converted to BOOLEAN. Any attempt to use the Boolean operators AND, NOT, OR and IS will fail. NOT 'False', for example, is invalid.

A BOOLEAN can be explicitly converted to and from string with CAST. UNKNOWN is not available for any
form of casting.

**Other Notes**

- The type is represented in the API with the `FB_BOOLEAN` type and `FB_TRUE` and `FB_FALSE` constants.
- The value `TRUE` is greater than the value `FALSE`.

### 3.7. Binary Data Types

BLOBs (Binary Large Objects) are complex structures used to store text and binary data of an undefined length, often very large.

**Syntax**

```
BLOB [SUB_TYPE <subtype>]
[SEGMENT SIZE <segment size>]
[CHARACTER SET <character set>]
[COLLATE <collation name>]
```

**Shortened syntax**

```
BLOB (<segment size>)
BLOB (<segment size>, <subtype>)
BLOB (, <subtype>)
```

Formally, the COLLATE clause is not part of the data type declaration, and its position depends on the syntax of the statement.

**Segment Size**

Specifying the BLOB segment size is throwback to times past, when applications for working with BLOB data were written in C (Embedded SQL) with the help of the `gpre` pre-compiler. Nowadays, it is effectively irrelevant. The segment size for BLOB data is determined by the client side and is usually larger than the data page size, in any case.

### 3.7.1. BLOB Subtypes

The optional `SUB_TYPE` parameter specifies the nature of data written to the column. Firebird provides two pre-defined subtypes for storing user data:

**Subtype 0: BINARY**

If a subtype is not specified, the specification is assumed to be for untyped data and the default `SUB_TYPE 0` is applied. The alias for subtype zero is `BINARY`. This is the subtype to specify when the data are any form of binary file or stream: images, audio, word-processor files, PDFs and so on.
Subtype 1: TEXT

Subtype 1 has an alias, TEXT, which can be used in declarations and definitions. For instance, BLOB SUB_TYPE TEXT. It is a specialized subtype used to store plain text data that is too large to fit into a string type. A CHARACTER SET may be specified, if the field is to store text with a different encoding to that specified for the database. From Firebird 2.0, a COLLATE clause is also supported.

Specifying a CHARACTER SET without SUB_TYPE implies SUB_TYPE TEXT.

Custom Subtypes

It is also possible to add custom data subtypes, for which the range of enumeration from -1 to -32,768 is reserved. Custom subtypes enumerated with positive numbers are not allowed, as the Firebird engine uses the numbers from 2-upward for some internal subtypes in metadata.

3.7.2. BLOB Specifics

Size

The maximum size of a BLOB field is limited to 4GB, regardless of whether the server is 32-bit or 64-bit. (The internal structures related to BLOBs maintain their own 4-byte counters.) For a page size of 4 KB (4096 bytes) the maximum size is lower — slightly less than 2GB.

Operations and Expressions

Text BLOBs of any length and any character set — including multi-byte — can be operands for practically any statement or internal functions. The following operators are supported completely:

\[
= \quad \text{(assignment)}
\]
\[
=, <>, <, \leq, >, \geq \quad \text{(comparison)}
\]
\[
|| \quad \text{(concatenation)}
\]
\[
\text{BETWEEN, IS [NOT] DISTINCT FROM,}
\]
\[
\text{IN, ANY | SOME,}
\]
\[
\text{ALL}
\]

Partial support:

- An error occurs with these if the search argument is larger than or equal to 32 KB:

\[
\text{STARTING [WITH], LIKE, CONTAINING}
\]

- Aggregation clauses work not on the contents of the field itself, but on the BLOB ID. Aside from that, there are some quirks:

\[
\text{SELECT DISTINCT}
\]
\[
\text{ORDER BY}
\]
\[
\text{returns several NULL values by mistake if they are present}
\]
GROUP BY concatenates the same strings if they are adjacent to each other, but does not do it if they are remote from each other.

**BLOB Storage**

- By default, a regular record is created for each BLOB and it is stored on a data page that is allocated for it. If the entire BLOB fits onto this page, it is called a *level 0 BLOB*. The number of this special record is stored in the table record and occupies 8 bytes.
- If a BLOB does not fit onto one data page, its contents are put onto separate pages allocated exclusively to it (blob pages), while the numbers of these pages are stored into the BLOB record. This is a *level 1 BLOB*.
- If the array of page numbers containing the BLOB data does not fit onto a data page, the array is put on separate blob pages, while the numbers of these pages are put into the BLOB record. This is a *level 2 BLOB*.
- Levels higher than 2 are not supported.

*See also*

FILTER, DECLARE FILTER

### 3.7.3. ARRAY Type

The support of arrays in the Firebird DBMS is a departure from the traditional relational model. Supporting arrays in the DBMS could make it easier to solve some data-processing tasks involving large sets of similar data.

Arrays in Firebird are stored in BLOB of a specialized type. Arrays can be one-dimensional and multi-dimensional and of any data type except BLOB and ARRAY.

**Example**

```sql
CREATE TABLE SAMPLE_ARR (
    ID INTEGER NOT NULL PRIMARY KEY,
    ARR_INT INTEGER [4]
);
```

This example will create a table with a field of the array type consisting of four integers. The subscripts of this array are from 1 to 4.

**Specifying Explicit Boundaries for Dimensions**

By default, dimensions are 1-based—subscripts are numbered from 1. To specify explicit upper and lower bounds of the subscript values, use the following syntax:

```
'[' <lower>:<upper> ']
```
Adding More Dimensions

A new dimension is added using a comma in the syntax. In this example we create a table with a two-dimensional array, with the lower bound of subscripts in both dimensions starting from zero:

```sql
CREATE TABLE SAMPLE_ARR2 (
    ID INTEGER NOT NULL PRIMARY KEY,
    ARR_INT INTEGER [0:3, 0:3]
);
```

The DBMS does not offer much in the way of language or tools for working with the contents of arrays. The database employee.fdb, found in the `../examples/empbuild` directory of any Firebird distribution package, contains a sample stored procedure showing some simple work with arrays:

**PSQL Source for `SHOW_LANGS`, a procedure involving an array**

```sql
CREATE OR ALTER PROCEDURE SHOW_LANGS (
    CODE VARCHAR(5),
    GRADE SMALLINT,
    CTY VARCHAR(15))
RETURNS (LANGUAGES VARCHAR(15))
AS
    DECLARE VARIABLE I INTEGER;
BEGIN
    I = 1;
    WHILE (I <= 5) DO
    BEGIN
        SELECT LANGUAGE_REQ[:I]
        FROM JOB
        WHERE (JOB_CODE = :CODE)
        AND (JOB_GRADE = :GRADE)
        AND (JOB_COUNTRY = :CTY)
        AND (LANGUAGE_REQ IS NOT NULL)
        INTO :LANGUAGES;

        IF (LANGUAGES = '') THEN
            /* PRINTS 'NULL' INSTEAD OF BLANKS */
            LANGUAGES = 'NULL';
            I = I +1;
            SUSPEND;
        END
    END
END
```

If the features described are enough for your tasks, you might consider using arrays in your projects. Currently, no improvements are planned to enhance support for arrays in Firebird.
3.8. Special Data Types

“Special” data types ...

3.8.1. SQL_NULL Data Type

The SQL_NULL type holds no data, but only a state: NULL or NOT NULL. It is not available as a data type for declaring table fields, PSQL variables or parameter descriptions. It was added to support the use of untyped parameters in expressions involving the IS NULL predicate.

An evaluation problem occurs when optional filters are used to write queries of the following type:

```
WHERE col1 = :param1 OR :param1 IS NULL
```

After processing, at the API level, the query will look like this:

```
WHERE col1 = ? OR ? IS NULL
```

This is a case where the developer writes an SQL query and considers :param1 as though it were a variable that he can refer to twice. However, at the API level, the query contains two separate and independent _parameters_. The server cannot determine the type of the second parameter since it comes in association with IS NULL.

The SQL_NULL data type solves this problem. Whenever the engine encounters an “? IS NULL” predicate in a query, it assigns the SQL_NULL type to the parameter, which will indicate that parameter is only about “nullness” and the data type or the value need not be addressed.

The following example demonstrates its use in practice. It assumes two named parameters — say, :size and :colour — which might, for example, get values from on-screen text fields or drop-down lists. Each named parameter corresponds with two positional parameters in the query.

```
SELECT
  SH.SIZE, SH.COLOUR, SH.PRICE
FROM SHIRTS SH
WHERE (SH.SIZE = ? OR ? IS NULL)
  AND (SH.COLOUR = ? OR ? IS NULL)
```

Explaining what happens here assumes the reader is familiar with the Firebird API and the passing of parameters in XSQLVAR structures — what happens under the surface will not be of interest to those who are not writing drivers or applications that communicate using the “naked” API.

The application passes the parameterized query to the server in the usual positional ?-form. Pairs of “identical” parameters cannot be merged into one so, for two optional filters, for example, four positional parameters are needed: one for each ? in our example.

After the call to isc_dsql_describe_bind(), the SQLTYPE of the second and fourth parameters will be
set to SQL_NULL. Firebird has no knowledge of their special relation with the first and third parameters: that responsibility lies entirely on the application side.

Once the values for size and colour have been set (or left unset) by the user and the query is about to be executed, each pair of XSQLVARs must be filled as follows:

**User has supplied a value**

First parameter (value compare): set *sqldata to the supplied value and *sqlind to 0 (for NOT NULL)

Second parameter (NULL test): set sqldata to null (null pointer, not SQL NULL) and *sqlind to 0 (for NOT NULL)

**User has left the field blank**

Both parameters: set sqldata to null (null pointer, not SQL NULL) and *sqlind to -1 (indicating NULL)

In other words: The value compare parameter is always set as usual. The SQL_NULL parameter is set the same, except that sqldata remains null at all times.

### 3.9. Conversion of Data Types

When composing an expression or specifying an operation, the aim should be to use compatible data types for the operands. When a need arises to use a mixture of data types, it should prompt you to look for a way to convert incompatible operands before subjecting them to the operation. The ability to convert data may well be an issue if you are working with Dialect 1 data.

#### 3.9.1. Explicit Data Type Conversion

The CAST function enables explicit conversion between many pairs of data types.

**Syntax**

```sql
CAST (<expression> AS <target_type>)
```

```plaintext
<target_type> ::= <domain_or_non_array_type> | <array_datatype>

<domain_or_non_array_type> ::= !! See Scalar Data Types Syntax !!

<array_datatype> ::= !! See Array Data Types Syntax !!
```

See also CAST() in Chapter Built-in Scalar Functions.

**Casting to a Domain**

When you cast to a domain, any constraints declared for it are taken into account, i.e., NOT NULL or CHECK constraints. If the value does not pass the check, the cast will fail.
If `TYPE OF` is additionally specified — casting to its base type — any domain constraints are ignored during the cast. If `TYPE OF` is used with a character type (CHAR/VARCHAR), the character set and collation are retained.

**Casting to `TYPE OF COLUMN`**

When operands are cast to the type of a column, the specified column may be from a table or a view.

Only the type of the column itself is used. For character types, the cast includes the character set, but not the collation. The constraints and default values of the source column are not applied.

**Example**

```sql
CREATE TABLE TTT (     S VARCHAR (40) CHARACTER SET UTF8 COLLATE UNICODE_CI_AI );
COMMIT;

SELECT CAST ('I have many friends' AS TYPE OF COLUMN TTT.S) FROM RDB$DATABASE;
```

**Conversions Possible for the `CAST` Function**

*Table 8. Conversions with CAST*

<table>
<thead>
<tr>
<th>From Data Type</th>
<th>To Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric types</td>
<td>Numeric types, [VAR]CHAR, BLOB</td>
</tr>
<tr>
<td>[VAR]CHAR</td>
<td>[VAR]CHAR, BLOB, Numeric types, DATE, TIME, TIMESTAMP, BOOLEAN</td>
</tr>
<tr>
<td>BLOB</td>
<td>[VAR]CHAR, BLOB, Numeric types, DATE, TIME, TIMESTAMP, BOOLEAN</td>
</tr>
<tr>
<td>DATE, TIME</td>
<td>[VAR]CHAR, BLOB, TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>[VAR]CHAR, BLOB, DATE, TIME</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BOOLEAN, [VAR]CHAR, BLOB</td>
</tr>
</tbody>
</table>

To convert string data types to the BOOLEAN type, the value must be (case-insensitive) 'true' or 'false', or NULL.

⚠️ Keep in mind that partial information loss is possible. For instance, when you cast the `TIMESTAMP` data type to the `DATE` data type, the time-part is lost.

**Literal Formats**

To cast string data types to the DATE, TIME or TIMESTAMP data types, you need the string argument to be one of the predefined date and time literals (see Table 9) or a representation of the date in one of the allowed *date-time literal* formats:
```plaintext
<timestamp_format> ::= 
{ [YYYY<p>]MM<p>DD[<p>HH[<p>mm[<p>SS[<p>NNNN]]]]] 
| MM<p>DD[<p>YYYY[<p>HH[<p>mm[<p>SS[<p>NNNN]]]]]] 
| DD<p>MM[<p>YYYY[<p>HH[<p>mm[<p>SS[<p>NNNN]]]]]] 
| MM<p>DD[<p>YY[<p>HH[<p>mm[<p>SS[<p>NNNN]]]]]] 
| DD<p>MM[<p>YY[<p>HH[<p>mm[<p>SS[<p>NNNN]]]]]] 
| NOW 
| TODAY 
| TOMORROW 
| YESTERDAY } 

<date_format> ::= 
{ [YYYY<p>]MM<p>DD 
| MM<p>DD[<p>YYYY] 
| DD<p>MM[<p>YYYY] 
| MM<p>DD[<p>YY] 
| DD<p>MM[<p>YY] 
| TODAY 
| TOMORROW 
| YESTERDAY } 

<time_format> ::= 
{ [HH<p>mm[<p>SS[<p>NNNN]]] 
| NOW } 

<p> ::= whitespace | . | : | , | - | /
```

**Table 9. Date and Time Literal Format Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp_format</td>
<td>Format of timestamp literal</td>
</tr>
<tr>
<td>date_literal</td>
<td>Format of date literal</td>
</tr>
<tr>
<td>time_literal</td>
<td>Format of time literal</td>
</tr>
<tr>
<td>YYYYY</td>
<td>Four-digit year</td>
</tr>
<tr>
<td>YY</td>
<td>Two-digit year</td>
</tr>
<tr>
<td>MM</td>
<td>Month. It may contain 1 or 2 digits (1-12 or 01-12). You can also specify the three-letter shorthand name or the full name of a month in English. Case-insensitive</td>
</tr>
<tr>
<td>DD</td>
<td>Day. It may contain 1 or 2 digits (1-31 or 01-31)</td>
</tr>
<tr>
<td>HH</td>
<td>Hour. It may contain 1 or 2 digits (0-23 or 00-23)</td>
</tr>
<tr>
<td>mm</td>
<td>Minutes. It may contain 1 or 2 digits (0-59 or 00-59)</td>
</tr>
<tr>
<td>SS</td>
<td>Seconds. It may contain 1 or 2 digits (0-59 or 00-59)</td>
</tr>
<tr>
<td>NNNN</td>
<td>Ten-thousandths of a second. It may contain from 1 to 4 digits (0-9999)</td>
</tr>
</tbody>
</table>
Table 10. Literals with Predefined Values of Date and Time

<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'NOW'</td>
<td>Current date and time</td>
<td>DATE</td>
</tr>
<tr>
<td>'TODAY'</td>
<td>Current date</td>
<td>DATE with zero time</td>
</tr>
<tr>
<td>'TOMORROW'</td>
<td>Current date + 1 (day)</td>
<td>DATE with zero time</td>
</tr>
<tr>
<td>'YESTERDAY'</td>
<td>Current date - 1 (day)</td>
<td>DATE with zero time</td>
</tr>
</tbody>
</table>

Use of the complete specification of the year in the four-digit form — YYYY — is strongly recommended, to avoid confusion in date calculations and aggregations.

Sample Date Literal Interpretations

```sql
select
cast('04.12.2014' as date) as d1, -- DD.MM.YYYY
cast('04 12 2014' as date) as d2, -- MM DD YYYY
cast('4-12-2014' as date) as d3, -- MM-DD-YYYY
cast('04/12/2014' as date) as d4, -- MM/DD/YYYY
cast('04,12,2014' as date) as d5, -- MM,DD,YYYY
cast('04.12.14' as date) as d6, -- DD.MM.YY
  -- DD.MM with current year

cast('04.12.' as date) as d7,
  -- MM/DD with current year

cast('04/12' as date) as d8,

cast('2014/12/04' as date) as d9, -- YYYY/MM/DD

cast('2014 12 04' as date) as d10, -- YYYY MM DD

cast('2014.12.04' as date) as d11, -- YYYY.MM.DD

cast('2014-12-04' as date) as d12, -- YYYY-MM-DD

cast('4 Jan 2014' as date) as d13, -- DD MM YYYY

cast('2014 Jan 4' as date) as d14, -- YYYY MM DD

cast('Jan 4, 2014' as date) as d15, -- MM DD, YYYY

cast('11:37' as time) as t1, -- HH:mm

cast('11:37:12' as time) as t2, -- HH:mm:ss

cast('11:37:12.1234' as time) as t3, -- HH:mm:ss.nnnn

cast('11.37.12' as time) as t4, -- HH.mm.ss
  -- DD.MM.YYYY HH:mm

cast('04.12.2014 11:37' as timestamp) as dt1,
  -- MM/DD/YYYY HH:mm:ss

cast('04/12/2014 11:37:12' as timestamp) as dt2,
  -- DD.MM.YYYY HH:mm:ss

cast('04.12.2014 11:31:12.1234' as timestamp) as dt3,
  -- MM/DD/YYYY HH:mm:ss

cast('04/12/2014 11.37.12' as timestamp) as dt4
```
Shorthand Casts for Date and Time Data Types

Firebird allows the use of a shorthand “C-style” type syntax for casts from string to the types DATE, TIME and TIMESTAMP. The SQL standard calls these datetime literals.

Syntax

```sql
<data_type> 'date_literal_string'
```

Example

```
-- 1
UPDATE PEOPLE
SET AGECAT = 'SENIOR'
WHERE BIRTHDATE < DATE '1-Jan-1943';
-- 2
INSERT INTO APPOINTMENTS
(EMPLOYEE_ID, CLIENT_ID, APP_DATE, APP_TIME)
VALUES (973, 8804, DATE 'today' + 2, TIME '16:00');
-- 3
NEW.LASTMOD = TIMESTAMP 'now';
```

These shorthand expressions are evaluated directly during parsing, as though the statement were already prepared for execution. Thus, even if the query is run several times, the value of, for instance, `timestamp 'now'` remains the same no matter how much time passes.

If you need the time to be evaluated at each execution, use the full `CAST` syntax. An example of using such an expression in a trigger:

```
NEW.CHANGE_DATE = CAST('now' AS TIMESTAMP);
```

Firebird 4 will no longer allow these implicit datetime values like 'now', 'today', etc in these shorthand casts. It is advisable to switch to using the full `CAST` expression for implicit values.

3.9.2. Implicit Data Type Conversion

Implicit data conversion is not possible in Dialect 3 — the `CAST` function is almost always required to avoid data type clashes.

In Dialect 1, in many expressions, one type is implicitly cast to another without the need to use the `CAST` function. For instance, the following statement in Dialect 1 is valid:
and the date literal will be cast to the date type implicitly.

In Dialect 3, this statement will throw error 35544569, “Dynamic SQL Error: expression evaluation not supported, Strings cannot be added or subtracted in dialect 3” — a cast will be needed:

```sql
UPDATE ATABLE
SET ADATE = '25.12.2016' + 1
```

or, with the short cast:

```sql
UPDATE ATABLE
SET ADATE = DATE '25.12.2016' + 1
```

In Dialect 1, mixing integer data and numeric strings is usually possible because the parser will try to cast the string implicitly. For example,

```sql
2 + '1'
```

will be executed correctly.

In Dialect 3, an expression like this will raise an error, so you will need to write it as a CAST expression:

```sql
2 + CAST('1' AS SMALLINT)
```

The exception to the rule is during string concatenation.

**Implicit Conversion During String Concatenation**

When multiple data elements are being concatenated, all non-string data will undergo implicit conversion to string, if possible.

*Example*

```sql
SELECT 30||' days hath September, April, June and November' CONCAT$
FROM RDB$DATABASE;
```

```
CONCAT$
--------------------------
30 days hath September, April, June and November
```
3.10. Custom Data Types — Domains

In Firebird, the concept of a “user-defined data type” is implemented in the form of the *domain*. Creating a domain does not truly create a new data type, of course. A domain provides the means to encapsulate an existing data type with a set of attributes and make this “capsule” available for multiple usage across the whole database. If several tables need columns defined with identical or nearly identical attributes, a domain makes sense.

Domain usage is not limited to column definitions for tables and views. Domains can be used to declare input and output parameters and variables in PSQL code.

3.10.1. Domain Attributes

A domain definition contains required and optional attributes. The *data type* is a required attribute. Optional attributes include:

- a default value
- to allow or forbid NULL
- CHECK constraints
- character set (for character data types and text BLOB fields)
- collation (for character data types)

*Sample domain definition*

```sql
CREATE DOMAIN BOOL3 AS SMALLINT
   CHECK (VALUE IS NULL OR VALUE IN (0, 1));
```

*See also*

Explicit Data Type Conversion for the description of differences in the data conversion mechanism when domains are specified for the TYPE OF and TYPE OF COLUMN modifiers.

3.10.2. Domain Override

While defining a column using a domain, it is possible to override some of the attributes inherited from the domain. Table 3.9 summarises the rules for domain override.

*Table 11. Rules for Overriding Domain Attributes in Column Definition*

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Override?</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Default value</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Text character set</td>
<td>Yes</td>
<td>It can be also used to restore the default database values for the column</td>
</tr>
<tr>
<td>Text collation sequence</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
### 3.10.3. Creating and Administering Domains

A domain is created with the DDL statement `CREATE DOMAIN`.

**Short Syntax**

```
CREATE DOMAIN name [AS] <type> 
[DEFAULT {<const> | <literal> | NULL | <context_var>}]
[NOT NULL] [CHECK (<condition>)]
[COLLATE <collation>]
```

See also
`CREATE DOMAIN` in the Data Definition Language (DDL) section.

**Altering a Domain**

To change the attributes of a domain, use the DDL statement `ALTER DOMAIN`. With this statement you can:

- rename the domain
- change the data type
- drop the current default value
- set a new default value
- drop the NOT NULL constraint
- set the NOT NULL constraint
- drop an existing CHECK constraint
- add a new CHECK constraint

**Short Syntax**

```
ALTER DOMAIN name 
[{{TO new_name}}]
[{{SET DEFAULT { <literal> | NULL | <context_var> } | DROP DEFAULT}}]
[{{SET | DROP} NOT NULL ]
[{{ADD [CONSTRAINT] CHECK (<dom_condition>) | DROP CONSTRAINT}}]
```
When changing a domain, its dependencies must be taken into account: whether there are table columns, any variables, input and/or output parameters with the type of this domain declared in the PSQL code. If you change domains in haste, without carefully checking them, your code may stop working!

When you convert data types in a domain, you must not perform any conversions that may result in data loss. Also, for example, if you convert VARCHAR to INTEGER, check carefully that all data using this domain can be successfully converted.

See also
ALTER DOMAIN in the Data Definition Language (DDL) section.

Deletng (Dropping) a Domain

The DDL statement DROP DOMAIN deletes a domain from the database, provided it is not in use by any other database objects.

Syntax

DROP DOMAIN name

Any user connected to the database can delete a domain.

Example

DROP DOMAIN Test_Domain

See also
DROP DOMAIN in the Data Definition Language (DDL) section.

3.11. Data Type Declaration Syntax

This section documents the syntax of declaring data types. Data type declaration most commonly occurs in DDL statements, but also in CAST and EXECUTE BLOCK.

The syntax documented below is referenced from other parts of this language reference.

3.11.1. Scalar Data Types Syntax

The scalar data types are simple data types that hold a single value. For reasons of organisation, the
syntax of BLOB types are defined separately in BLOB Data Types Syntax.

Scalar Data Types Syntax

$$<\text{domain_or_non_array_type}> ::=$$
$$<\text{scalar_datatype}>$$
$$| <\text{blob_datatype}>$$
$$| [\text{TYPE OF}] \text{domain}$$
$$| \text{TYPE OF COLUMN rel.col}$$

$$<\text{scalar_datatype}> ::=$$
$$\text{SMALLINT} | \text{INT[EGER]} | \text{BIGINT}$$
$$| \text{FLOAT} | \text{DOUBLE PRECISION}$$
$$| \text{BOOLEAN}$$
$$| \text{DATE} | \text{TIME} | \text{TIMESTAMP}$$
$$| \{\text{DECIMAL} | \text{NUMERIC}\} [(\text{precision} [, \text{scale}])]$$
$$| \{\text{VARCHAR} | \{\text{CHAR} | \text{CHARACTER}\} \text{VARYING}\} (\text{length})$$
$$[\text{CHARACTER SET charset}]$$
$$| \{\text{CHAR} | \text{CHARACTER}\} [(\text{length})] [\text{CHARACTER SET charset}]$$
$$| \{\text{NCHAR} | \text{NATIONAL} \{\text{CHARACTER} | \text{CHAR}\}\} \text{VARYING} (\text{length})$$
$$| \{\text{NCHAR} | \text{NATIONAL} \{\text{CHARACTER} | \text{CHAR}\}\} [(\text{length})]$$

Table 12. Arguments for the Scalar Data Types Syntax

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain</td>
<td>Domain (only non-array domains)</td>
</tr>
<tr>
<td>rel</td>
<td>Name of a table or view</td>
</tr>
<tr>
<td>col</td>
<td>Name of a column in a table or view (only columns of a non-array type)</td>
</tr>
<tr>
<td>precision</td>
<td>Numeric precision in decimal digits. From 1 to 18</td>
</tr>
<tr>
<td>scale</td>
<td>Scale, or number of decimals. From 0 to 18. It must be less than or equal to precision</td>
</tr>
<tr>
<td>length</td>
<td>The maximum length of a string, in characters</td>
</tr>
<tr>
<td>charset</td>
<td>Character set</td>
</tr>
<tr>
<td>domain_or_non_array_type</td>
<td>Non-array types that can be used in PSQL code and casts</td>
</tr>
</tbody>
</table>

Use of Domains in Declarations

A domain name can be specified as the type of a PSQL parameter or local variable. The parameter or variable will inherit all domain attributes. If a default value is specified for the parameter or variable, it overrides the default value specified in the domain definition.

If the TYPE OF clause is added before the domain name, only the data type of the domain is used: any of the other attributes of the domain—NOT NULL constraint, CHECK constraints, default value—are neither checked nor used. However, if the domain is of a text type, its character set and collation sequence are always used.
Use of Column Type in Declarations

Input and output parameters or local variables can also be declared using the data type of columns in existing tables and views. The `TYPE OF COLUMN` clause is used for that, specifying `relationname .columnname` as its argument.

When `TYPE OF COLUMN` is used, the parameter or variable inherits only the data type and — for string types — the character set and collation sequence. The constraints and default value of the column are ignored.

3.11.2. BLOB Data Types Syntax

The BLOB data types hold binary, character or custom format data of unspecified size. For more information, see Binary Data Types.

**BLOB Data Types Syntax**

```
<blob_datatype> ::= 
    BLOB [SUB_TYPE {subtype_num | subtype_name}] 
    [SEGMENT SIZE seglen] [CHARACTER SET charset] 
    | BLOB [(seglen [, subtype_num])] 
```

**Table 13. Arguments for the Blob Data Types Syntax**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charset</td>
<td>Character set (ignored for sub-types other than TEXT/1)</td>
</tr>
<tr>
<td>subtype_num</td>
<td>BLOB subtype number</td>
</tr>
<tr>
<td>subtype_name</td>
<td>BLOB subtype mnemonic name; this can be TEXT, BINARY, or one of the (other) standard or custom names defined in RDB$TYPES for RDB$FIELD_NAME = 'RDB$FIELD_SUB_TYPE'.</td>
</tr>
<tr>
<td>seglen</td>
<td>Segment size, cannot be greater than 65,535, defaults to 80 when not specified. See also Segment Size</td>
</tr>
</tbody>
</table>

3.11.3. Array Data Types Syntax

The array data types hold multiple scalar values in a single or multi-dimensional array. For more information, see ARRAY Type

**Array Data Types Syntax**

```
<array_datatype> ::= 
    {SMALLINT | INT[EGER] | BIGINT} <array_dim> 
    | {FLOAT | DOUBLE PRECISION} <array_dim> 
    | BOOLEAN <array_dim> 
    | {DATE | TIME | TIMESTAMP} <array_dim> 
    | {DECIMAL | NUMERIC} [(precision [, scale])] <array_dim> 
    | {VARCHAR | {CHAR | CHARACTER} VARYING} (length) 
    <array_dim> [CHARACTER SET charset] 
```
Table 14. Arguments for the Array Data Types Syntax

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_dim</td>
<td>Array dimensions</td>
</tr>
<tr>
<td>precision</td>
<td>Numeric precision in decimal digits. From 1 to 18</td>
</tr>
<tr>
<td>scale</td>
<td>Scale, or number of decimals. From 0 to 18. It must be less than or equal to precision</td>
</tr>
<tr>
<td>length</td>
<td>The maximum length of a string, in characters; optional for fixed-width character types, defaults to 1</td>
</tr>
<tr>
<td>charset</td>
<td>Character set</td>
</tr>
<tr>
<td>m, n</td>
<td>Integer numbers defining the index range of an array dimension</td>
</tr>
</tbody>
</table>
Chapter 4. Common Language Elements

This chapter covers the elements that are common throughout the implementation of the SQL language — the expressions that are used to extract and operate on conditions about data and the predicates that test the truth of those assertions.

4.1. Expressions

SQL expressions provide formal methods for evaluating, transforming and comparing values. SQL expressions may include table columns, variables, constants, literals, various statements and predicates and also other expressions. The complete list of possible tokens in expressions follows.

Description of Expression Elements

Column name
Identifier of a column from a specified table used in evaluations or as a search condition. A column of the array type cannot be an element in an expression except when used with the IS [NOT] NULL predicate.

Array element
An expression may contain a reference to an array member i.e., <array_name>[s], where s is the subscript of the member in the array <array_name>

Arithmetic operators
The +, -, *, / characters used to calculate values

Concatenation operator
The || (“double-pipe”) operator used to concatenate strings

Logical operators
The reserved words NOT, AND and OR, used to combine simple search conditions in order to create complex conditions

Comparison operators
The symbols =, <>, !=, ^=, ^=, <, <=, >, >=, !<, ^<, !>, ^> and ^>

Comparison predicates
LIKE, STARTING WITH, CONTAINING, SIMILAR TO, BETWEEN, IS [NOT] NULL, IS [NOT] {TRUE | FALSE | UNKNOWN} and IS [NOT] DISTINCT FROM

Existential predicates
Predicates used to check the existence of values in a set. The IN predicate can be used both with sets of comma-separated constants and with subqueries that return a single column. The EXISTS, SINGULAR, ALL, ANY and SOME predicates can be used only with subqueries.

Constant or Literal
Numbers, or string literals enclosed in apostrophes, Boolean values TRUE, FALSE and UNKNOWN, NULL
Date/time literal
An expression, similar to a string literal enclosed in apostrophes, that can be interpreted as a date, time or timestamp value. Date literals can be predefined literals ('TODAY', 'NOW', etc.) or strings of characters and numerals, such as '25.12.2016 15:30:35', that can be resolved as date and/or time strings.

Context variable
An internally-defined context variable

Local variable
Declared local variable, input or output parameter of a PSQL module (stored procedure, trigger, unnamed PSQL block in DSQL)

Positional parameter
A member of in an ordered group of one or more unnamed parameters passed to a stored procedure or prepared query

Subquery
A SELECT statement enclosed in parentheses that returns a single (scalar) value or, when used in existential predicates, a set of values

Function identifier
The identifier of an internal or external function in a function expression

Type cast
An expression explicitly converting data of one data type to another using the CAST function (CAST (<value> AS <datatype>)). For date/time literals only, the shorthand syntax <datatype> <value> is also supported (DATE '2016-12-25').

Conditional expression
Expressions using CASE and related internal functions

Parentheses
Bracket pairs (... used to group expressions. Operations inside the parentheses are performed before operations outside them. When nested parentheses are used, the most deeply nested expressions are evaluated first and then the evaluations move outward through the levels of nesting.

COLLATE clause
Clause applied to CHAR and VARCHAR types to specify the character-set-specific collation sequence to use in string comparisons

NEXT VALUE FOR sequence
Expression for obtaining the next value of a specified generator (sequence). The internal GEN_ID() function does the same.
4.1.1. Literals (Constants)

A literal — or constant — is a value that is supplied directly in an SQL statement, not derived from an expression, a parameter, a column reference nor a variable. It can be a string or a number.

**String Literals**

A string literal is a series of characters enclosed between a pair of apostrophes (“single quotes”). The maximum length of a string literal is 32,765 for `CHAR/VARCHAR`, or 65,533 bytes for `BLOB`; the maximum character count will be determined by the number of bytes used to encode each character.

- Double quotes are *NOT VALID* for quoting strings. The SQL standard reserves double quotes for a different purpose: quoting identifiers.
- If a literal apostrophe is required within a string constant, it is “escaped” by prefixing it with another apostrophe. For example, 'Mother O''Reilly's homemade hooch'.
- Care should be taken with the string length if the value is to be written to a `CHAR` or `VARCHAR` column. The maximum length for a `CHAR` or `VARCHAR` literal is 32,765 bytes.

The character set of a string constant is assumed to be the same as the character set of its destined storage.

**String Literals in Hexadecimal Notation**

From Firebird 2.5 forward, string literals can be entered in hexadecimal notation, so-called “binary strings”. Each pair of hex digits defines one byte in the string. Strings entered this way will have character set OCTETS by default, but the *introducer syntax* can be used to force a string to be interpreted as another character set.

**Syntax**

```
{x|X}'<hexstring>'
```

```
<hexstring> ::= an even number of <hexdigit>
<hexdigit> ::= one of 0..9, A..F, a..f
```

**Examples**

```
select x'4E657276656E' from rdb$database
-- returns 4E657276656E, a 6-byte 'binary' string

select _ascii x'4E657276656E' from rdb$database
-- returns 'Nerven' (same string, now interpreted as ASCII text)

select _iso8859_1 x'53E46765' from rdb$database
-- returns 'Säge' (4 chars, 4 bytes)
```
Notes

The client interface determines how binary strings are displayed to the user. The `isql` utility, for example, uses upper case letters A-F, while FlameRobin uses lower case letters. Other client programs may use other conventions, such as displaying spaces between the byte pairs: '4E 65 72 76 65 6E'.

The hexadecimal notation allows any byte value (including 00) to be inserted at any position in the string. However, if you want to coerce it to anything other than OCTETS, it is your responsibility to supply the bytes in a sequence that is valid for the target character set.

Alternative String Literals

Since Firebird 3.0, it is possible to use a character, or character pair, other than the doubled (escaped) apostrophe, to embed a quoted string inside another string. The keyword `q` or `Q` preceding a quoted string informs the parser that certain left-right pairs or pairs of identical characters within the string are the delimiters of the embedded string literal.

Syntax

```
<alternative string literal> ::= { q | Q } <quote> <start char> [<char> ...] <end char> <quote>
```

Rules

When `<start char>` is ‘(’, ‘{’, ‘[’ or ‘<’, `<end char>` is paired up with its respective “partner”, viz. ‘)’, ‘}’, ‘]’ and ‘>’. In other cases, `<end char>` is the same as `<start char>`.

Inside the string, i.e. `<char>` items, single (not escaped) quotes can be used. Each quote will be part of the result string.

Examples

```
select q'{abc{def}ghi}' from rdb$database;        -- result: abc{def}ghi
select q'!That's a string!' from rdb$database;    -- result: That's a string
```

Introducer Syntax for String Literals

If necessary, a string literal may be preceded by a character set name, itself prefixed with an underscore “_”. This is known as *introducer syntax*. Its purpose is to inform the engine about how to interpret and store the incoming string.

Example
Number Literals

A number literal is any valid number in a supported notation:

- In SQL, for numbers in the standard decimal notation, the decimal point is always represented by period character (', full-stop, dot); thousands are not separated. Inclusion of commas, blanks, etc. will cause errors.
- Exponential notation is supported. For example, 0.0000234 can be expressed as 2.34e-5.
- Hexadecimal notation is supported by Firebird 2.5 and higher versions — see below.

The format of the literal decides the type (<d> for a decimal digit, <h> for a hexadecimal digit):

<table>
<thead>
<tr>
<th>Format</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;d&gt;[&lt;d&gt; ...]</td>
<td>INTEGER or BIGINT (depends on if value fits in the type)</td>
</tr>
<tr>
<td>0{x</td>
<td>X} &lt;h&gt;&lt;h&gt;[&lt;h&gt;&lt;h&gt; ...]</td>
</tr>
<tr>
<td>&lt;d&gt;[&lt;d&gt; ...] &quot;.&quot; [&lt;d&gt; ...]</td>
<td>NUMERIC(18, n) where n depends on the number of digits after the decimal point</td>
</tr>
<tr>
<td>&lt;d&gt;[&lt;d&gt; ...][&quot;.&quot; [&lt;d&gt; ...]] E &lt;d&gt;[&lt;d&gt; ...]</td>
<td>DOUBLE PRECISION</td>
</tr>
</tbody>
</table>

Hexadecimal Notation for Numbers

From Firebird 2.5 forward, integer values can be entered in hexadecimal notation. Numbers with 1-8 hex digits will be interpreted as type INTEGER; numbers with 9-16 hex digits as type BIGINT.

Syntax

\[0{x|X}<\text{hexdigits}>\]

\(<\text{hexdigits}> ::= 1-16\ of\ <\text{hexdigit}>\)

\(<\text{hexdigit}> ::= \text{one of 0..9, A..F, a..f}\)

Examples

- select 0x6FAA00D3 from rdb$database -- returns 117088467
- select 0x4F9 from rdb$database -- returns 1273
- select 0x6E44F9A8 from rdb$database -- returns 1850014120
- select 0x9E44F9A8 from rdb$database -- returns -1639646808 (an INTEGER)
- select 0x09E44F9A8 from rdb$database -- returns 2655320488 (a BIGINT)
- select 0x28ED678A4C987 from rdb$database -- returns 720001751632263
Hexadecimal Value Ranges

• Hex numbers in the range 0 .. 7FFF FFFF are positive INTEGERs with values between 0 .. 2147483647 decimal. To coerce a number to BIGINT, prepend enough zeroes to bring the total number of hex digits to nine or above. That changes the type but not the value.

• Hex numbers between 8000 0000 .. FFFF FFFF require some attention:
  ◦ When written with eight hex digits, as in 0x9E44F9A8, a value is interpreted as 32-bit INTEGER. Since the leftmost bit (sign bit) is set, it maps to the negative range -2147483648 .. -1 decimal.
  ◦ With one or more zeroes prepended, as in 0x09E44F9A8, a value is interpreted as 64-bit BIGINT in the range 0000 0000 8000 0000 .. 0000 0000 FFFF FFFF. The sign bit is not set now, so they map to the positive range 2147483648 .. 4294967295 decimal.

Thus, in this range — and only in this range — prepending a mathematically insignificant 0 results in a totally different value. This is something to be aware of.

• Hex numbers between 1 0000 0000 .. 7FFF FFFF FFFF FFFF are all positive BIGINT.

• Hex numbers between 8000 0000 0000 0000 .. FFFF FFFF FFFF FFFF are all negative BIGINT.

• A SMALLINT cannot be written in hex, strictly speaking, since even 0x1 is evaluated as INTEGER. However, if you write a positive integer within the 16-bit range 0x0000 (decimal zero) to 0x7FFF (decimal 32767) it will be converted to SMALLINT transparently.

It is possible to write to a negative SMALLINT in hex, using a 4-byte hex number within the range 0xFFFF8000 (decimal -32768) to 0xFFFFFFFF (decimal -1).

Boolean Literals

A Boolean literal is one of TRUE, FALSE or UNKNOWN.

4.1.2. SQL Operators

SQL operators comprise operators for comparing, calculating, evaluating and concatenating values.

Operator Precedence

SQL Operators are divided into four types. Each operator type has a precedence, a ranking that determines the order in which operators and the values obtained with their help are evaluated in an expression. The higher the precedence of the operator type is, the earlier it will be evaluated. Each operator has its own precedence within its type, that determines the order in which they are evaluated in an expression.

Operators with the same precedence are evaluated from left to right. To force a different evaluation order, operations can be grouped by means of parentheses.

Table 15. Operator Type Precedence
<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Precedence</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td>1</td>
<td>Strings are concatenated before any other operations take place</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>2</td>
<td>Arithmetic operations are performed after strings are concatenated, but before comparison and logical operations</td>
</tr>
<tr>
<td>Comparison</td>
<td>3</td>
<td>Comparison operations take place after string concatenation and arithmetic operations, but before logical operations</td>
</tr>
<tr>
<td>Logical</td>
<td>4</td>
<td>Logical operators are executed after all other types of operators</td>
</tr>
</tbody>
</table>

**Concatenation Operator**

The concatenation operator, two pipe characters known as “double pipe” — `||` — concatenates (connects together) two character strings to form a single string. Character strings can be constants or values obtained from columns or other expressions.

**Example**

```
SELECT LAST_NAME || ', ' || FIRST_NAME AS FULL_NAME
FROM EMPLOYEE
```

**Arithmetic Operators**

**Table 16. Arithmetic Operator Precedence**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>+signed_number</td>
<td>Unary plus</td>
<td>1</td>
</tr>
<tr>
<td>-signed_number</td>
<td>Unary minus</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>2</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>3</td>
</tr>
</tbody>
</table>

**Example**

```
UPDATE T
    SET A = 4 + 1/(B-C)*D
```

Where operators have the same precedence, they are evaluated in left-to-right sequence.
### Comparison Operators

#### Table 17. Comparison Operator Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>Checks that the expression on the left is (not) NULL or the Boolean value on the right</td>
<td>1</td>
</tr>
<tr>
<td>=</td>
<td>Is equal to, is identical to</td>
<td>2</td>
</tr>
<tr>
<td>&lt;&gt; , != , ^=</td>
<td>Is not equal to</td>
<td>2</td>
</tr>
<tr>
<td>&gt;</td>
<td>Is greater than</td>
<td>2</td>
</tr>
<tr>
<td>&lt;</td>
<td>Is less than</td>
<td>2</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Is greater than or equal to</td>
<td>2</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Is less than or equal to</td>
<td>2</td>
</tr>
<tr>
<td>!&gt;, =~ , ^&gt;</td>
<td>Is not greater than</td>
<td>2</td>
</tr>
<tr>
<td>!&lt; , ~&lt; , ^&lt;</td>
<td>Is not less than</td>
<td>2</td>
</tr>
</tbody>
</table>

This group also includes comparison predicates BETWEEN, LIKE, CONTAINING, SIMILAR TO and others.

**Example**

```sql
IF (SALARY > 1400) THEN ...
```

**See also**

[Other Comparison Predicates](#).

### Logical Operators

#### Table 18. Logical Operator Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>Negation of a search condition</td>
<td>1</td>
</tr>
<tr>
<td>AND</td>
<td>Combines two or more predicates, each of which must be true for the entire predicate to be true</td>
<td>2</td>
</tr>
<tr>
<td>OR</td>
<td>Combines two or more predicates, of which at least one predicate must be true for the entire predicate to be true</td>
<td>3</td>
</tr>
</tbody>
</table>

**Example**

```sql
IF (A < B OR (A > C AND A > D) AND NOT (C = D)) THEN ...
```

**NEXT VALUE FOR**

[Available in](#)
NEXT VALUE FOR returns the next value of a sequence. SEQUENCE is the SQL-standard term for what is historically called a generator in Firebird and its ancestor, InterBase. The NEXT VALUE FOR operator is equivalent to the legacy GEN_ID (…, 1) function, and is the recommended syntax for retrieving the next sequence value.

Unlike GEN_ID (…, 1), the NEXT VALUE FOR variant does not take any parameters and thus, provides no way to retrieve the current value of a sequence, nor to step the next value by more than 1. GEN_ID (…, <step value>) is still needed for these tasks. A step value of 0 returns the current sequence value.

Example

NEW.CUST_ID = NEXT VALUE FOR CUSTSEQ;

See also

SEQUENCE (GENERATOR), GEN_ID()

4.1.3. Conditional Expressions

A conditional expression is one that returns different values according to how a certain condition is met. It is composed by applying a conditional function construct, of which Firebird supports several. This section describes only one conditional expression construct: CASE. All other conditional expressions apply internal functions derived from CASE and are described in Conditional Functions.

CASE

Available in

DSQL, PSQL

The CASE construct returns a single value from a number of possible values. Two syntactic variants are supported:

- The simple CASE, comparable to a case construct in Pascal or a switch in C
- The searched CASE, which works like a series of “if ... else if ... else if” clauses.

Simple CASE

Syntax

... 
CASE <test-expr> 
  WHEN <expr> THEN <result>
When this variant is used, test-expr is compared to the first expr, second expr and so on, until a match is found, and the corresponding result is returned. If no match is found, defaultresult from the optional ELSE clause is returned. If there are no matches and no ELSE clause, NULL is returned.

The matching works identically to the “=” operator. That is, if test-expr is NULL, it does not match any expr, not even an expression that resolves to NULL.

The returned result does not have to be a literal value: it might be a field or variable name, compound expression or NULL literal.

Example

```
SELECT
    NAME,
    AGE,
    CASE UPPER(SEX)
        WHEN 'M' THEN 'Male'
        WHEN 'F' THEN 'Female'
        ELSE 'Unknown'
    END GENDER,
    RELIGION
FROM PEOPLE
```

A short form of the simple CASE construct is the DECODE function.

Searched CASE

Syntax

```
CASE
    WHEN <bool_expr> THEN <result>
    [WHEN <bool_expr> THEN <result> …]
    [ELSE <defaultresult>] END
```

The bool_expr expression is one that gives a ternary logical result: TRUE, FALSE or NULL. The first expression to return TRUE determines the result. If no expressions return TRUE, defaultresult from the optional ELSE clause is returned as the result. If no expressions return TRUE and there is no ELSE clause, the result will be NULL.

As with the simple CASE construct, the result need not be a literal value: it might be a field or variable name, a compound expression, or be NULL.
Example

```sql
CANVOTE = CASE
    WHEN AGE >= 18 THEN 'Yes'
    WHEN AGE < 18 THEN 'No'
    ELSE 'Unsure'
END
```

### 4.1.4. NULL in Expressions

NULL is not a value in SQL, but a state indicating that the value of the element either is *unknown* or it does not exist. It is not a zero, nor a void, nor an “empty string”, and it does not act like any value.

When you use NULL in numeric, string or date/time expressions, the result will always be NULL. When you use NULL in logical (Boolean) expressions, the result will depend on the type of the operation and on other participating values. When you compare a value to NULL, the result will be *unknown*.

![Warning](image)

NULL means NULL but, in Firebird, the logical result *unknown* is also represented by NULL.

**Expressions Returning NULL**

Expressions in this list will always return NULL:

- `1 + 2 + 3 + NULL`
- `'Home ' || 'sweet ' || NULL`
- `MyField = NULL`
- `MyField <> NULL`
- `NULL = NULL`
- `not (NULL)`

If it seems difficult to understand why, remember that NULL is a state that stands for “unknown”.

**NULL in Logical Expressions**

It has already been shown that `NOT (NULL)` results in NULL. The interaction is a bit more complicated for the logical AND and logical OR operators:

- `NULL or false → NULL`
- `NULL or true → true`
- `NULL or NULL → NULL`
- `NULL and false → false`
- `NULL and true → NULL`
- `NULL and NULL → NULL`

![Tip](image)

As a basic rule-of-thumb, if applying `TRUE` instead of `NULL` produces a different result than applying `FALSE`, then the outcome of the original expression is
Examples

(1 = NULL) or (1 <> 1) -- returns NULL
(1 = NULL) or FALSE -- returns NULL
(1 = NULL) or (1 = 1) -- returns TRUE
(1 = NULL) or TRUE -- returns TRUE
(1 = NULL) or (1 = NULL) -- returns NULL
(1 = NULL) or UNKNOWN -- returns NULL
(1 = NULL) and (1 <> 1) -- returns FALSE
(1 = NULL) and FALSE -- returns FALSE
(1 = NULL) and (1 = 1) -- returns NULL
(1 = NULL) and TRUE -- returns NULL
(1 = NULL) and (1 = NULL) -- returns NULL
(1 = NULL) and UNKNOWN -- returns NULL

4.1.5. Subqueries

A subquery is a special form of expression that is actually a query embedded within another query. Subqueries are written in the same way as regular SELECT queries, but they must be enclosed in parentheses. Subquery expressions can be used in the following ways:

• To specify an output column in the SELECT list
• To obtain values or conditions for search predicates (the WHERE, HAVING clauses).
• To produce a set that the enclosing query can select from, as though were a regular table or view. Subqueries like this appear in the FROM clause (derived tables) or in a Common Table Expression (CTE)

Correlated Subqueries

A subquery can be correlated. A query is correlated when the subquery and the main query are interdependent. To process each record in the subquery, it is necessary to fetch a record in the main query; i.e. the subquery fully depends on the main query.

Sample Correlated Subquery

```
SELECT *
FROM Customers C
WHERE EXISTS
  (SELECT *
     FROM Orders O
     WHERE C.cnum = O.cnum
     AND O.adate = DATE '10.03.1990');
```

When subqueries are used to get the values of the output column in the SELECT list, a subquery must return a scalar result (see below).
Scalar Results

Subqueries used in search predicates, other than existential and quantified predicates, must return a **scalar** result; that is, not more than one column from not more than one matching row or aggregation. If the result would return more, a run-time error will occur (“Multiple rows in a singleton select...”).

Although it is reporting a genuine error, the message can be slightly misleading. A “singleton SELECT” is a query that must not be capable of returning more than one row. However, “singleton” and “scalar” are not synonymous: not all singleton SELECTs are required to be scalar; and single-column selects can return multiple rows for existential and quantified predicates.

Subquery Examples

1. A subquery as the output column in a **SELECT** list:

```sql
SELECT 
  e.first_name, 
  e.last_name, 
  (SELECT 
    sh.new_salary 
  FROM 
    salary_history sh 
  WHERE 
    sh.emp_no = e.emp_no 
  ORDER BY sh.change_date DESC ROWS 1) AS last_salary 
FROM 
  employee e
```

2. A subquery in the **WHERE** clause for obtaining the employee’s maximum salary and filtering by it:

```sql
SELECT 
  e.first_name, 
  e.last_name, 
  e.salary 
FROM employee e 
WHERE 
  e.salary = ( 
    SELECT MAX(ie.salary) 
    FROM employee ie 
  )
```

4.2. Predicates

A predicate is a simple expression asserting some fact, let’s call it $P$. If $P$ resolves as TRUE, it succeeds. If it resolves to FALSE or NULL (UNKNOWN), it fails. A trap lies here, though: suppose the predicate, $P$, returns FALSE. In this case $\text{NOT}(P)$ will return TRUE. On the other hand, if $P$ returns
NULL (unknown), then $\text{NOT}(P)$ returns NULL as well.

In SQL, predicates can appear in CHECK constraints, WHERE and HAVING clauses, CASE expressions, the IIF() function and in the ON condition of JOIN clauses, and—since Firebird 3.0—anywhere a normal expression can occur.

### 4.2.1. Conditions

A condition —or Boolean expression—is a statement about the data that, like a predicate, can resolve to TRUE, FALSE or NULL. Conditions consist of one or more predicates, possibly negated using $\text{NOT}$ and connected by $\text{AND}$ and $\text{OR}$ operators. Parentheses may be used for grouping predicates and controlling evaluation order.

A predicate may embed other predicates. Evaluation sequence is in the outward direction, i.e., the innermost predicates are evaluated first. Each “level” is evaluated in precedence order until the truth value of the ultimate condition is resolved.

### 4.2.2. Comparison Predicates

A comparison predicate consists of two expressions connected with a comparison operator. There are six traditional comparison operators:

\[ =, >, <, \geq, \leq, \neq \]

For the complete list of comparison operators with their variant forms, see [Comparison Operators](#).

If one of the sides (left or right) of a comparison predicate has NULL in it, the value of the predicate will be UNKNOWN.

**Examples**

1. Retrieve information about computers with the CPU frequency not less than 500 MHz and the price lower than $800:

   ```sql
   SELECT *
   FROM Pc
   WHERE speed >= 500 AND price < 800;
   ```

2. Retrieve information about all dot matrix printers that cost less than $300:

   ```sql
   SELECT *
   FROM Printer
   WHERE ptrtype = 'matrix' AND price < 300;
   ```

3. The following query will return no data, even if there are printers with no type specified for them, because a predicate that compares NULL with NULL returns NULL:
SELECT *
FROM Printer
WHERE ptrtype = NULL AND price < 300;

On the other hand, ptrtype can be tested for NULL and return a result: it is just that it is not a *comparison* test:

SELECT *
FROM Printer
WHERE ptrtype IS NULL AND price < 300;

— see **IS [NOT] NULL**.

---

**Note about String Comparison**

When CHAR and VARCHAR fields are compared for equality, trailing spaces are ignored in all cases.

---

**Other Comparison Predicates**

Other comparison predicates are marked by keyword symbols.

**BETWEEN**

*Available in*

DSQL, PSQL, ESQL

*Syntax*

```
<value> [NOT] BETWEEN <value_1> AND <value_2>
```

The BETWEEN predicate tests whether a value falls within a specified range of two values. (NOT BETWEEN tests whether the value does not fall within that range.)

The operands for BETWEEN predicate are two arguments of compatible data types. Unlike in some other DBMS, the BETWEEN predicate in Firebird is not symmetrical — if the lower value is not the first argument, the BETWEEN predicate will always return FALSE. The search is inclusive (the values represented by both arguments are included in the search). In other words, the BETWEEN predicate could be rewritten:

```
<value> >= <value_1> AND <value> <= <value_2>
```

When BETWEEN is used in the search conditions of DML queries, the Firebird optimizer can use an index on the searched column, if it is available.
Example

```sql
SELECT *
FROM EMPLOYEE
WHERE HIRE_DATE BETWEEN date '1992-01-01' AND CURRENT_DATE
```

LIKE

Available in
DSQL, PSQL, ESQL

Syntax

```sql
<match_value> [NOT] LIKE <pattern>
[ESCAPE <escape character>]
```

- `<match_value>` ::= character-type expression
- `<pattern>` ::= search pattern
- `<escape character>` ::= escape character

The LIKE predicate compares the character-type expression with the pattern defined in the second expression. Case- or accent-sensitivity for the comparison is determined by the collation that is in use. A collation can be specified for either operand, if required.

Wildcards

Two wildcard symbols are available for use in the search pattern:

- the percentage symbol (%) will match any sequence of zero or more characters in the tested value
- the underscore character (_) will match any single character in the tested value

If the tested value matches the pattern, taking into account wildcard symbols, the predicate is TRUE.

Using the ESCAPE Character Option

If the search string contains either of the wildcard symbols, the ESCAPE clause can be used to specify an escape character. The escape character must precede the '%' or '_' symbol in the search string, to indicate that the symbol is to be interpreted as a literal character.

Examples using LIKE

1. Find the numbers of departments whose names start with the word “Software”:

   ```sql
   SELECT DEPT_NO
   FROM DEPT
   WHERE DEPT_NAME LIKE 'Software%';
   ```
It is possible to use an index on the DEPT_NAME field if it exists.

**About LIKE and the Optimizer**

Actually, the LIKE predicate does not use an index. However, if the predicate takes the form of LIKE 'string%', it will be converted to the STARTING WITH predicate, which will use an index. This optimization only works for literal patterns, not for parameters.

So, if you need to search for the beginning of a string, it is recommended to use the STARTING WITH predicate instead of the LIKE predicate.

2. Search for employees whose names consist of 5 letters, start with the letters “Sm” and end with “th”. The predicate will be true for such names as “Smith” and “Smyth”.

```sql
SELECT first_name
FROM employee
WHERE first_name LIKE 'Sm_th'
```

3. Search for all clients whose address contains the string “Rostov”:

```sql
SELECT *
FROM CUSTOMER
WHERE ADDRESS LIKE '%Rostov%
```

If you need to do a case-insensitive search for something enclosed inside a string (LIKE '%Abc%'), use of the CONTAINING predicate is recommended, in preference to the LIKE predicate.

4. Search for tables containing the underscore character in their names. The ‘#’ character is used as the escape character:

```sql
SELECT RDB$RELATION_NAME
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME LIKE '%#_%' ESCAPE '#'
```

See also

STARTING WITH, CONTAINING, SIMILAR TO

STARTING WITH

Available in

DSQL, PSQL, ESQL
The STARTING WITH predicate searches for a string or a string-like type that starts with the characters in its `value` argument. The case- and accent-sensitivity of STARTING WITH depends on the collation of the first `value`.

When STARTING WITH is used in the search conditions of DML queries, the Firebird optimizer can use an index on the searched column, if it exists.

**Example**

Search for employees whose last names start with “Jo”:

```
SELECT LAST_NAME, FIRST_NAME
FROM EMPLOYEE
WHERE LAST_NAME STARTING WITH 'Jo'
```

**See also**

LIKE
CONTAINING

**Available in**

DSQL, PSQL, ESQL

---

The CONTAINING predicate searches for a string or a string-like type looking for the sequence of characters that matches its argument. It can be used for an alphanumeric (string-like) search on numbers and dates. A CONTAINING search is not case-sensitive. However, if an accent-sensitive collation is in use then the search will be accent-sensitive.

**Examples**

1. Search for projects whose names contain the substring “Map”:

   ```
   SELECT *
   FROM PROJECT
   WHERE PROJ_NAME CONTAINING 'Map';
   ```

   Two rows with the names “AutoMap” and “MapBrowser port” are returned.

2. Search for changes in salaries with the date containing number 84 (in this case, it means changes that took place in 1984):
SELECT *  
FROM SALARY_HISTORY  
WHERE CHANGE_DATE CONTAINING 84;

See also
LIKE
SIMILAR TO
Available in
DSQL, PSQL

Syntax

```
string-expression [NOT] SIMILAR TO <pattern> [ESCAPE <escape-char>]
```

- `<pattern>` ::= an SQL regular expression
- `<escape-char>` ::= a single character

SIMILAR TO matches a string against an SQL regular expression pattern. Unlike in some other languages, the pattern must match the entire string in order to succeed — matching a substring is not enough. If any operand is NULL, the result is NULL. Otherwise, the result is TRUE or FALSE.

Syntax: SQL Regular Expressions

The following syntax defines the SQL regular expression format. It is a complete and correct top-down definition. It is also highly formal, rather long and probably perfectly fit to discourage everybody who hasn't already some experience with regular expressions (or with highly formal, rather long top-down definitions). Feel free to skip it and read the next section, Building Regular Expressions, which uses a bottom-up approach, aimed at the rest of us.

```
<regular expression> ::= <regular term> ['|' <regular term> ...]
<regular term> ::= <regular factor> ...
<regular factor> ::= <regular primary> [<quantifier>]
<quantifier> ::= ? | * | + | '{' <m> [',' <n>] '}'
<m>, <n> ::= unsigned int, with <m> <= <n> if both present
<regular primary> ::=  
  <character> | <character class> | %  
  | ( <regular expression> )
<character> ::= <escaped character> | <non-escaped character>
<escaped character> ::=
Building Regular Expressions

In this section are the elements and rules for building SQL regular expressions.

Characters

Within regular expressions, most characters represent themselves. The only exceptions are the special characters below:

\[ ( ) | ^ - + * \_ ? \} \]

... and the escape character, if it is defined.

A regular expression that contains no special character or escape characters matches only strings that are identical to itself (subject to the collation in use). That is, it functions just like the ‘=’ operator:

'Apple' similar to 'Apple' -- true
'Apples' similar to 'Apple' -- false
'Apple' similar to 'Apples' -- false
'APPLE' similar to 'Apple' -- depends on collation

Wildcards

The known SQL wildcards ‘\_’ and ‘%’ match any single character and a string of any length, respectively:
Notice how ‘%’ also matches the empty string.

**Character Classes**

A bunch of characters enclosed in brackets define a character class. A character in the string matches a class in the pattern if the character is a member of the class:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Citroen' similar to 'Cit[arju]oen'</td>
<td>false</td>
</tr>
<tr>
<td>'Citroen' similar to 'Ci[tr]oen'</td>
<td>true</td>
</tr>
<tr>
<td>'Citroen' similar to 'Ci[tr][tr]oen'</td>
<td>true</td>
</tr>
</tbody>
</table>

As can be seen from the second line, the class only matches a single character, not a sequence.

Within a class definition, two characters connected by a hyphen define a range. A range comprises the two endpoints and all the characters that lie between them in the active collation. Ranges can be placed anywhere in the class definition without special delimiters to keep them apart from the other elements.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Datte' similar to 'Dat[q-u]e'</td>
<td>true</td>
</tr>
<tr>
<td>'Datte' similar to 'Dat[abq-uy]e'</td>
<td>true</td>
</tr>
<tr>
<td>'Datte' similar to 'Dat[bcg-km-pwz]e'</td>
<td>false</td>
</tr>
</tbody>
</table>

**Predefined Character Classes**

The following predefined character classes can also be used in a class definition:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[:ALPHA:]</td>
<td>Latin letters a..z and A..Z. With an accent-insensitive collation, this class also matches accented forms of these characters.</td>
</tr>
<tr>
<td>[:DIGIT:]</td>
<td>Decimal digits 0..9.</td>
</tr>
<tr>
<td>[:ALNUM:]</td>
<td>Union of [:ALPHA:] and [:DIGIT:].</td>
</tr>
<tr>
<td>[:UPPER:]</td>
<td>Uppercase Latin letters A..Z. Also matches lowercase with case-insensitive collation and accented forms with accent-insensitive collation.</td>
</tr>
</tbody>
</table>
Lowercase Latin letters a..z. Also matches uppercase with case-insensitive collation and accented forms with accent-insensitive collation.

Matches the space character (ASCII 32).

Matches horizontal tab (ASCII 9), linefeed (ASCII 10), vertical tab (ASCII 11), formfeed (ASCII 12), carriage return (ASCII 13) and space (ASCII 32).

Including a predefined class has the same effect as including all its members. Predefined classes are only allowed within class definitions. If you need to match against a predefined class and nothing more, place an extra pair of brackets around it.

'Erdbeere' similar to 'Erd[[:ALNUM:]]eere' -- true
'Erdbeere' similar to 'Erd[[:DIGIT:]]eere' -- false
'Erdbeere' similar to 'Erd[a[:SPACE:]]b]eere' -- true
'Erdbeere' similar to [[:ALPHA:]] -- false
'E' similar to [[:ALPHA:]] -- true

If a class definition starts with a caret, everything that follows is excluded from the class. All other characters match:

'Framboise' similar to 'Fra[^ck-p]boise' -- false
'Framboise' similar to 'Fr[^a]^[^a]boise' -- false
'Framboise' similar to 'Fra[^[:DIGIT:]]boise' -- true

If the caret is not placed at the start of the sequence, the class contains everything before the caret, except for the elements that also occur after the caret:

'Grapefruit' similar to 'Grap[a-m^f-i]fruit' -- true
'Grapefruit' similar to 'Grap[abc^xyz]fruit' -- false
'Grapefruit' similar to 'Grap[abc^de]fruit' -- false
'Grapefruit' similar to 'Grap[abe^de]fruit' -- false
'3' similar to '[:DIGIT:]^4-8' -- true
'6' similar to '[:DIGIT:]^4-8' -- false

Lastly, the already mentioned wildcard ‘_’ is a character class of its own, matching any single character.

**Quantifiers**

A question mark (?) immediately following a character or class indicates that the preceding item may occur 0 or 1 times in order to match:
### An asterisk (*) immediately following a character or class indicates that the preceding item may occur 0 or more times in order to match:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Icaque' similar to 'Ica*que'</td>
<td>true</td>
</tr>
<tr>
<td>'Icaque' similar to 'Icar*que'</td>
<td>true</td>
</tr>
<tr>
<td>'Icaque' similar to 'I[a-c]*que'</td>
<td>true</td>
</tr>
<tr>
<td>'Icaque' similar to '*'</td>
<td>true</td>
</tr>
<tr>
<td>'Icaque' similar to '[[[:ALPHA:]]]*'</td>
<td>true</td>
</tr>
<tr>
<td>'Icaque' similar to 'Ica[xyz]*e'</td>
<td>false</td>
</tr>
</tbody>
</table>

### A plus sign (+) immediately following a character or class indicates that the preceding item must occur 1 or more times in order to match:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Jujube' similar to 'Ju_+'</td>
<td>true</td>
</tr>
<tr>
<td>'Jujube' similar to 'Ju+jube'</td>
<td>true</td>
</tr>
<tr>
<td>'Jujube' similar to 'Ju+uber+'</td>
<td>false</td>
</tr>
<tr>
<td>'Jujube' similar to 'J[jux]+be'</td>
<td>true</td>
</tr>
<tr>
<td>'Jujube' similar to 'J[[[:DIGIT:]]]+ujube'</td>
<td>false</td>
</tr>
</tbody>
</table>

### If a character or class is followed by a number enclosed in braces ('{ ' and ' }'), it must be repeated exactly that number of times in order to match:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Kiwi' similar to 'Ki{2}wi'</td>
<td>false</td>
</tr>
<tr>
<td>'Kiwi' similar to 'K[ipw]{2}i'</td>
<td>true</td>
</tr>
<tr>
<td>'Kiwi' similar to 'K[ipw]{2}'</td>
<td>false</td>
</tr>
<tr>
<td>'Kiwi' similar to 'K[ipw]{3}'</td>
<td>true</td>
</tr>
</tbody>
</table>

### If the number is followed by a comma (','), the item must be repeated at least that number of times in order to match:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Limone' similar to 'Li{2},mone'</td>
<td>false</td>
</tr>
<tr>
<td>'Limone' similar to 'Li{1},mone'</td>
<td>true</td>
</tr>
<tr>
<td>'Limone' similar to 'Li[nezom]{2,}'</td>
<td>true</td>
</tr>
</tbody>
</table>

### If the braces contain two numbers separated by a comma, the second number not smaller than the first, then the item must be repeated at least the first number and at most the second number of times in order to match:
The quantifiers ‘?’, ‘*’ and ‘+’ are shorthand for \{0,1\}, \{0,\} and \{1,\}, respectively.

**OR-ing Terms**

Regular expression terms can be OR'ed with the ‘|’ operator. A match is made when the argument string matches at least one of the terms:

- 'Nektarin' similar to 'Nek|tarin' -- false
- 'Nektarin' similar to 'Nektarin|Persika' -- true
- 'Nektarin' similar to 'M_+|N_+|P_+' -- true

**Subexpressions**

One or more parts of the regular expression can be grouped into subexpressions (also called subpatterns) by placing them between parentheses ('(' and ')'). A subexpression is a regular expression in its own right. It can contain all the elements allowed in a regular expression, and can also have quantifiers added to it.

- 'Orange' similar to 'O(ra|ri|ro)nge' -- true
- 'Orange' similar to 'O(r[a-e])+nge' -- true
- 'Orange' similar to 'O(ra){2,4}nge' -- false
- 'Orange' similar to 'O(r(an|in)g|rong)?e' -- true

**Escaping Special Characters**

In order to match against a character that is special in regular expressions, that character has to be escaped. There is no default escape character; rather, the user specifies one when needed:

- 'Peer (Poire)' similar to 'P[^ ]+ \(P[^ ]+\)' escape '\' -- true
- 'Pera [Pear]' similar to 'P[^ ]+ #[P[^ ]+#]' escape '#' -- true
- 'Päron-äppledryck' similar to 'P%$-ä%' escape '$' -- true
- 'Pärondryck' similar to 'P%--ä%' escape '-' -- false

The last line demonstrates that the escape character can also escape itself, if needed.

**IS [NOT] DISTINCT FROM**

*Available in*

DSQL, PSQL
Syntax

\(<\text{operand1}>\ \text{IS} \ [\text{NOT}] \ \text{DISTINCT} \ \text{FROM} \ <\text{operand2}>\)

Two operands are considered \textit{DISTINCT} (different) if they have a different value or if one of them is \texttt{NULL} and the other non-null. They are considered \textit{NOT DISTINCT} (equal) if they have the same value or if both of them are \texttt{NULL}.

\texttt{IS [NOT] DISTINCT FROM} always returns TRUE or FALSE and never UNKNOWN (NULL) (unknown value). Operators ‘\texttt{=}’ and ‘\texttt{<>}’, conversely, will return UNKNOWN (NULL) if one or both operands are \texttt{NULL}.

\textbf{Table 19. Results of Various Comparison Predicates}

<table>
<thead>
<tr>
<th>Operand values</th>
<th>Result of various predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\texttt{=}</td>
</tr>
<tr>
<td>Same value</td>
<td>TRUE</td>
</tr>
<tr>
<td>Different values</td>
<td>FALSE</td>
</tr>
<tr>
<td>Both NULL</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>One NULL, one non-NULL</td>
<td>UNKNOWN</td>
</tr>
</tbody>
</table>

\textbf{Examples}

```sql
SELECT ID, NAME, TEACHER
FROM COURSES
WHERE START_DAY IS NOT DISTINCT FROM END_DAY;
```

-- PSQL fragment

```sql
IF (NEW.JOB IS DISTINCT FROM OLD.JOB)
THEN POST_EVENT 'JOB_CHANGED';
```

\textbf{See also}

\texttt{IS [NOT] NULL}, \texttt{Boolean IS [NOT]}

\textbf{Boolean IS [NOT]}

\textit{Available in}

DSQL, PSQL

\textbf{Syntax}

\texttt{<value> IS [NOT] \{ TRUE | FALSE | UNKNOWN \}}

The \texttt{IS} predicate with \texttt{Boolean} literal values checks if the expression on the left side matches the \texttt{Boolean} value on the right side. The expression on the left side must be of type \texttt{BOOLEAN}, otherwise an exception is raised.
The IS [NOT] UNKNOWN is equivalent to IS [NOT] NULL.

The right side of the predicate only accepts the literals TRUE, FALSE and UNKNOWN (and NULL). It does not accept expressions.

Using the IS predicate with a Boolean data type

```
-- Checking FALSE value
SELECT * FROM TBOOL WHERE BVAL IS FALSE;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>BVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>&lt;false&gt;</td>
</tr>
</tbody>
</table>

```
-- Checking UNKNOWN value
SELECT * FROM TBOOL WHERE BVAL IS UNKNOWN;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>BVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

See also
IS [NOT] NULL

IS [NOT] NULL

Available in
DSQL, PSQL, ESQL

Syntax

```
<value> IS [NOT] NULL
```

Since NULL is not a value, these operators are not comparison operators. The IS [NOT] NULL predicate tests that the expression on the left side has a value (IS NOT NULL) or has no value (IS NULL).

Example

Search for sales entries that have no shipment date set for them:

```
SELECT * FROM SALES
WHERE SHIP_DATE IS NULL;
```

Note regarding the IS predicates

Up to and including Firebird 2.5, the IS predicates, like the other comparison predicates, do not have precedence over the others. In Firebird 3.0 and higher, these predicates take precedence above the others.
4.2.3. Existential Predicates

This group of predicates includes those that use subqueries to submit values for all kinds of assertions in search conditions. Existential predicates are so called because they use various methods to test for the existence or non-existence of some condition, returning TRUE if the existence or non-existence is confirmed or FALSE otherwise.

**EXISTS**

*Available in*

DSQL, PSQL, ESQL

**Syntax**

```
[NOT] EXISTS (<select_stmt>)
```

The EXISTS predicate uses a subquery expression as its argument. It returns TRUE if the subquery result would contain at least one row; otherwise it returns FALSE.

NOT EXISTS returns FALSE if the subquery result would contain at least one row; it returns TRUE otherwise.

The subquery can specify multiple columns, or SELECT *, because the evaluation is made on the number of rows that match its criteria, not on the data.

**Examples**

1. Find those employees who have projects.

   ```sql
   SELECT *
   FROM employee
   WHERE EXISTS(SELECT *
                FROM employee_project ep
                WHERE ep.emp_no = employee.emp_no)
   ```

2. Find those employees who have no projects.

   ```sql
   SELECT *
   FROM employee
   WHERE NOT EXISTS(SELECT *
                    FROM employee_project ep
                    WHERE ep.emp_no = employee.emp_no)
   ```

**IN**

*Available in*

DSQL, PSQL, ESQL
The IN predicate tests whether the value of the expression on the left side is present in the set of values specified on the right side. The set of values cannot have more than 1500 items. The IN predicate can be replaced with the following equivalent forms:

\[
<\text{value}> \ [\text{NOT}] \ \text{IN} \ \langle<\text{select}_{}\text{stmt}> \ | \ <\text{value}_{}\text{list}>\rangle
\]

\[
<\text{value}_{}\text{list}> ::\:= \ <\text{value}_{}\text{1}> \ [, \ <\text{value}_{}\text{2}> \ \ldots]\]

When the IN predicate is used in the search conditions of DML queries, the Firebird optimizer can use an index on the searched column, if a suitable one exists.

In its second form, the IN predicate tests whether the value of the expression on the left side is present—or not present, if NOT IN is used—in the result of the executed subquery on the right side.

The subquery must be specified to result in only one column, otherwise the error “count of column list and variable list do not match” will occur.

Queries specified using the IN predicate with a subquery can be replaced with a similar query using the EXISTS predicate. For instance, the following query:

```
SELECT
  model, speed, hd
FROM PC
WHERE
  model \text{IN} \ (SELECT model
    FROM product
    WHERE maker = 'A');
```

can be replaced with a similar one using the EXISTS predicate:

```
SELECT
  model, speed, hd
FROM PC
WHERE
  EXISTS (SELECT *
    FROM product
    WHERE maker = 'A'
    \text{AND} product.model = PC.model);
```

However, a query using NOT IN with a subquery does not always give the same result as its NOT EXISTS counterpart. The reason is that EXISTS always returns TRUE or FALSE, whereas IN returns
NULL in one of these two cases:

a. when the test value is NULL and the IN () list is not empty
b. when the test value has no match in the IN () list and at least one list element is NULL

It is in only these two cases that IN () will return NULL while the corresponding EXISTS predicate will return FALSE (‘no matching row found’). In a search or, for example, an IF (…) statement, both results mean “failure” and it makes no difference to the outcome.

But, for the same data, NOT IN () will return NULL, while NOT EXISTS will return TRUE, leading to opposite results.

As an example, suppose you have the following query:

```
-- Looking for people who were not born
-- on the same day as any famous New York citizen
SELECT P1.name AS NAME
FROM Personnel P1
WHERE P1.birthday NOT IN (SELECT C1.birthday
                           FROM Celebrities C1
                           WHERE C1.birthcity = 'New York');
```

Now, assume that the NY celebrities list is not empty and contains at least one NULL birthday. Then for every citizen who does not share his birthday with a NY celebrity, NOT IN will return NULL, because that is what IN does. The search condition is thereby not satisfied and the citizen will be left out of the SELECT result, which is wrong.

For citizens whose birthday does match with a celebrity's birthday, NOT IN will correctly return FALSE, so they will be left out too, and no rows will be returned.

If the not EXISTS form is used:

```
-- Looking for people who were not born
-- on the same day as any famous New York citizen
SELECT P1.name AS NAME
FROM Personnel P1
WHERE NOT EXISTS (SELECT *
                  FROM Celebrities C1
                  WHERE C1.birthcity = 'New York'
                  AND C1.birthday = P1.birthday);
```

non-matches will have a NOT EXISTS result of TRUE and their records will be in the result set.

If there is any chance of NULLs being encountered when searching for a non-match, you will want to use NOT EXISTS.

Examples of use

1. Find employees with the names “Pete”, “Ann” and “Roger”:
2. Find all computers that have models whose manufacturer starts with the letter “A”:

```
SELECT
    model, speed, hd
FROM PC
WHERE
    model IN (SELECT model
                FROM product
                WHERE maker STARTING WITH 'A');
```

See also

**EXISTS**

**SINGULAR**

*Available in*

DSQL, PSQL, ESQL

**Syntax**

```
[NOT] SINGULAR (<select_stmt>)
```

The SINGULAR predicate takes a subquery as its argument and evaluates it as TRUE if the subquery returns exactly one result row; otherwise the predicate is evaluated as FALSE. The subquery may list several output columns since the rows are not returned anyway. They are only tested for (singular) existence. For brevity, people usually specify ‘SELECT *’. The SINGULAR predicate can return only two values: TRUE or FALSE.

**Example**

Find those employees who have only one project.

```
SELECT *
FROM employee
WHERE SINGULAR(SELECT *
                FROM employee_project ep
                WHERE ep.emp_no = employee.emp_no)
```

### 4.2.4. Quantified Subquery Predicates

A quantifier is a logical operator that sets the number of objects for which this condition is true. It is not a numeric quantity, but a logical one that connects the condition with the full set of possible objects. Such predicates are based on logical universal and existential quantifiers that are
recognised in formal logic.

In subquery expressions, quantified predicates make it possible to compare separate values with the results of subqueries; they have the following common form:

\[ \text{<value expression> <comparison operator> <quantifier> <subquery>} \]

**ALL**

*Available in*

DSQL, PSQL, ESQL

*Syntax*

\[ \text{<value> <op> ALL (<select_stmt>)} \]

When the **ALL** quantifier is used, the predicate is TRUE if every value returned by the subquery satisfies the condition in the predicate of the main query.

*Example*

Show only those clients whose ratings are higher than the rating of every client in Paris.

```sql
SELECT c1.*
FROM Customers c1
WHERE c1.rating > ALL
    (SELECT c2.rating
     FROM Customers c2
     WHERE c2.city = 'Paris')
```

If the subquery returns an empty set, the predicate is TRUE for every left-side value, regardless of the operator. This may appear to be contradictory, because every left-side value will thus be considered both smaller and greater than, both equal to and unequal to, every element of the right-side stream.

Nevertheless, it aligns perfectly with formal logic: if the set is empty, the predicate is true 0 times, i.e. for every row in the set.

**ANY and SOME**

*Available in*

DSQL, PSQL, ESQL

*Syntax*

\[ \text{<value> <op> \{ANY | SOME\} (<select_stmt>)} \]
The quantifiers ANY and SOME are identical in their behaviour. Apparently, both are present in the SQL standard so that they could be used interchangeably in order to improve the readability of operators. When the ANY or the SOME quantifier is used, the predicate is TRUE if any of the values returned by the subquery satisfies the condition in the predicate of the main query. If the subquery would return no rows at all, the predicate is automatically considered as FALSE.

*Example*

Show only those clients whose ratings are higher than those of one or more clients in Rome.

```
SELECT *
FROM Customers
WHERE rating > ANY
  (SELECT rating
   FROM Customers
   WHERE city = 'Rome')
```
Chapter 5. Data Definition (DDL) Statements

DDL is the data definition language subset of Firebird’s SQL language. DDL statements are used to create, modify and delete database objects that have been created by users. When a DDL statement is committed, the metadata for the object are created, changed or deleted.

5.1. DATABASE

This section describes how to create a database, connect to an existing database, alter the file structure of a database and how to delete one. It also explains how to back up a database in two quite different ways and how to switch the database to the “copy-safe” mode for performing an external backup safely.

5.1.1. CREATE DATABASE

Used for
Creating a new database

Available in
DSQL, ESQL

Syntax

CREATE {DATABASE | SCHEMA} <filespec>
  [[<db_initial_option>] [<db_initial_option> ...]]
  [[<db_config_option>] [<db_config_option> ...]]

<db_initial_option> ::=  
  USER username  
  | PASSWORD 'password'
  | ROLE rolename  
  | PAGE_SIZE [=] size  
  | LENGTH [=] num [PAGE[S]]  
  | SET NAMES 'charset'

<db_config_option> ::=  
  DEFAULT CHARACTER SET default_charset  
  [COLLATION collation] -- not supported in ESQL  
  | <sec_file>
  | DIFFERENCE FILE 'diff_file' -- not supported in ESQL

<filespec> ::= "" [server_spec]{filepath | db_alias} ""

<server_spec> ::=  
  host[/{port | service}]:  
  | \host\  
  | <protocol>://[host::{port | service}]/

<protocol> ::= inet | inet4 | inet6 | wnet | xnet
Each \textit{db\_initial\_option} and \textit{db\_config\_option} can occur at most once, except \textit{sec\_file}, which can occur zero or more times.

\textbf{Table 20. CREATE DATABASE Statement Parameters}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filespec</td>
<td>File specification for primary database file</td>
</tr>
<tr>
<td>server_spec</td>
<td>Remote server specification. Some protocols require specifying a hostname. Optionally includes a port number or service name. Required if the database is created on a remote server.</td>
</tr>
<tr>
<td>filepath</td>
<td>Full path and file name including its extension. The file name must be specified according to the rules of the platform file system being used.</td>
</tr>
<tr>
<td>db_alias</td>
<td>Database alias previously created in the \texttt{databases.conf} file</td>
</tr>
<tr>
<td>host</td>
<td>Host name or IP address of the server where the database is to be created</td>
</tr>
<tr>
<td>port</td>
<td>The port number where the remote server is listening (parameter \texttt{RemoteServicePort} in \texttt{firebird.conf} file)</td>
</tr>
<tr>
<td>service</td>
<td>Service name. Must match the parameter value of \texttt{RemoteServiceName} in \texttt{firebird.conf} file</td>
</tr>
<tr>
<td>username</td>
<td>Username of the owner of the new database. It may consist of up to 31 characters. The username can optionally be enclosed in single or double quotes. When a username is enclosed in double quotes, it is case-sensitive following the rules for quoted identifiers. When enclosed in single quotes, it behaves as if the value was specified without quotes. The user must be an administrator or have the CREATE DATABASE privilege.</td>
</tr>
<tr>
<td>password</td>
<td>Password of the user as the database owner. When using the Legacy_Auth authentication plugin, only the first 8 characters are used. Case-sensitive</td>
</tr>
<tr>
<td>rolename</td>
<td>The name of the role whose rights should be taken into account when creating a database. The role name can be enclosed in single or double quotes. When the role name is enclosed in double quotes, it is case-sensitive following the rules for quoted identifiers. When enclosed in single quotes, it behaves as if the value was specified without quotes.</td>
</tr>
<tr>
<td>size</td>
<td>Page size for the database, in bytes. Possible values are 4096, 8192 and 16384. The default page size is 8192.</td>
</tr>
<tr>
<td>num</td>
<td>Maximum size of the primary database file, or a secondary file, in pages</td>
</tr>
<tr>
<td>charset</td>
<td>Specifies the character set of the connection available to a client connecting after the database is successfully created. Single quotes are required.</td>
</tr>
</tbody>
</table>
The `CREATE DATABASE` statement creates a new database. You can use `CREATE DATABASE` or `CREATE SCHEMA`. They are synonymous, but we recommend to always use `CREATE DATABASE` as this may change in a future version of Firebird.

A database may consist of one or several files. The first (main) file is called the **primary file**, subsequent files are called **secondary file(s)**.

### Multi-file Databases

Nowadays, multi-file databases are considered an anachronism. It made sense to use multi-file databases on old file systems where the size of any file is limited. For instance, you could not create a file larger than 4 GB on FAT32.

The primary file specification is the name of the database file and its extension with the full path to it according to the rules of the OS platform file system being used. The database file must not exist at the moment the database is being created. If it does exist, you will get an error message, and the database will not be created.

If the full path to the database is not specified, the database will be created in one of the system directories. The particular directory depends on the operating system. For this reason, unless you have a strong reason to prefer that situation, always specify either the absolute path or an alias, when creating a database.

### Using a Database Alias

You can use aliases instead of the full path to the primary database file. Aliases are defined in the `databases.conf` file in the following format:

```plaintext
alias = filepath
```

Executing a `CREATE DATABASE` statement requires special consideration in the client application or database driver. As a result, it is not always possible to execute a `CREATE DATABASE` statement. Some drivers provide other ways to create databases. For example, Jaybird provides the class `org.firebirdsql.management.FBManager` to programmatically create a database.

If necessary, you can always fallback to `isql` to create a database.
Creating a Database on a Remote Server

If you create a database on a remote server, you need to specify the remote server specification. The remote server specification depends on the protocol being used. If you use the TCP/IP protocol to create a database, the primary file specification should look like this:

```
host[/{port|service}]:{filepath | db_alias}
```

If you use the Named Pipes protocol to create a database on a Windows server, the primary file specification should look like this:

```
\\host\{filepath | db_alias}
```

Since Firebird 3.0, there is also a unified URL-like syntax for the remote server specification. In this syntax, the first part specifies the name of the protocol, then a host name or IP address, port number, and path of the primary database file, or an alias.

The following values can be specified as the protocol:

- **INET**
  - TCP/IP (first tries to connect using the IPv6 protocol, if it fails, then IPv4)
- **INET4**
  - TCP/IP v4 (since Firebird 3.0.1)
- **INET6**
  - TCP/IP v6 (since Firebird 3.0.1)
- **WNET**
  - NetBEUI or Named Pipes Protocol
- **XNET**
  - local protocol (does not include a host, port and service name)

```
<protocol>://[host[::{port | service}]/]{filepath | db_alias}
```

Optional Parameters for `CREATE DATABASE`

**USER and PASSWORD**

Clauses for specifying the username and the password, respectively, of an existing user in the security database (security3.fdb or whatever is configured in the `SecurityDatabase` configuration). You do not have to specify the username and password if the `ISC_USER` and `ISC_PASSWORD` environment variables are set. The user specified in the process of creating the database will be its owner. This will be important when considering database and object privileges.
ROLE

The `ROLE` clause specifies the name of the role (usually `RDB$ADMIN`), which will be taken into account when creating the database. The role must be assigned to the user in the applicable security database.

PAGE_SIZE

Clause for specifying the database page size. This size will be set for the primary file and all secondary files of the database. If you specify the database page size less than 4,096, it will be automatically rounded up to 4,096. Other values not equal to either 4,096, 8,192 or 16,384 will be changed to the closest smaller supported value. If the database page size is not specified, it is set to the default value of 8,192.

LENGTH

Clause specifying the maximum size of the primary or secondary database file, in pages. When a database is created, its primary and secondary files will occupy the minimum number of pages necessary to store the system data, regardless of the value specified in the `LENGTH` clause. The `LENGTH` value does not affect the size of the only (or last, in a multi-file database) file. The file will keep increasing its size automatically when necessary.

SET NAMES

Clause specifying the character set of the connection available after the database is successfully created. The character set `NONE` is used by default. Notice that the character set should be enclosed in a pair of apostrophes (single quotes).

DEFAULT CHARACTER SET

Clause specifying the default character set for creating data structures of string data types. Character sets are used for `CHAR`, `VARCHAR` and `BLOB SUB_TYPE TEXT` data types. The character set `NONE` is used by default. It is also possible to specify the default `COLLATION` for the default character set, making that collation sequence the default for the default character set. The default will be used for the entire database except where an alternative character set, with or without a specified collation, is used explicitly for a field, domain, variable, cast expression, etc.

STARTING AT

Clause that specifies the database page number at which the next secondary database file should start. When the previous file is completely filled with data according to the specified page number, the system will start adding new data to the next database file.

DIFFERENCE FILE

Clause specifying the path and name for the file delta that stores any mutations to the database file after it has been switched to the “copy-safe” mode by the `ALTER DATABASE BEGIN BACKUP` statement. For the detailed description of this clause, see `ALTER DATABASE`.

Specifying the Database Dialect

Databases are created in Dialect 3 by default. For the database to be created in SQL dialect 1, you will need to execute the statement `SET SQL DIALECT 1` from script or the client application, e.g. in `isql`, before the `CREATE DATABASE` statement.
Who Can Create a Database

The CREATE DATABASE statement can be executed by:

- Administrators
- Users with the CREATE DATABASE privilege

Examples Using CREATE DATABASE

1. Creating a database in Windows, located on disk D with a page size of 4,096. The owner of the database will be the user wizard. The database will be in Dialect 1, and will use WIN1251 as its default character set.

   ```
   SET SQL DIALECT 1;
   CREATE DATABASE 'D:\test.fdb'
   USER 'wizard' PASSWORD 'player'
   PAGE_SIZE = 4096 DEFAULT CHARACTER SET WIN1251;
   ```

2. Creating a database in the Linux operating system with a page size of 8,192 (default). The owner of the database will be the user wizard. The database will be in Dialect 3 and will use UTF8 as its default character set, with UNICODE_CI_AI as the default collation.

   ```
   CREATE DATABASE '/home/firebird/test.fdb'
   USER 'wizard' PASSWORD 'player'
   DEFAULT CHARACTER SET UTF8 COLLATION UNICODE_CI_AI;
   ```

3. Creating a database on the remote server “baseserver” with the path specified in the alias “test” that has been defined previously in the file databases.conf. The TCP/IP protocol is used. The owner of the database will be the user wizard. The database will be in Dialect 3 and will use UTF8 as its default character set.

   ```
   CREATE DATABASE 'baseserver:test'
   USER 'wizard' PASSWORD 'player'
   DEFAULT CHARACTER SET UTF8;
   ```

4. Creating a database in Dialect 3 with UTF8 as its default character set. The primary file will contain up to 10,000 pages with a page size of 8,192. As soon as the primary file has reached the maximum number of pages, Firebird will start allocating pages to the secondary file test.fdb2. If that file is filled up to its maximum as well, test.fdb3 becomes the recipient of all new page allocations. As the last file, it has no page limit imposed on it by Firebird. New allocations will continue for as long as the file system allows it or until the storage device runs out of free space. If a LENGTH parameter were supplied for this last file, it would be ignored.

   ```
   SET SQL DIALECT 3;
   CREATE DATABASE 'baseserver:D:\test.fdb'
   USER 'wizard' PASSWORD 'player'
   ```
5. Creating a database in Dialect 3 with UTF8 as its default character set. The primary file will contain up to 10,000 pages with a page size of 8,192. As far as file size and the use of secondary files are concerned, this database will behave exactly like the one in the previous example.

```sql
SET SQL DIALECT 3;
CREATE DATABASE 'baseserver:D:\test.fdb'
USER 'wizard' PASSWORD 'player'
PAGE_SIZE = 8192
LENGTH 10000 PAGES
DEFAULT CHARACTER SET UTF8
FILE 'D:\test.fdb2'
FILE 'D:\test.fdb3'
STARTING AT PAGE 20001;
```

See also

`ALTER DATABASE`, `DROP DATABASE`

### 5.1.2. ALTER DATABASE

*Used for*

Altering the file organisation of a database, toggling its “copy-safe” state, managing encryption, and other database-wide configuration

*Available in*

DSQL, ESQL — limited feature set

*Syntax*

```sql
ALTER {DATABASE | SCHEMA} <alter_db_option> [<alter_db_option> ...]

<alter_db_option> ::=  
  <add_sec_clause>  
  | {ADD DIFFERENCE FILE 'diff_file' | DROP DIFFERENCE FILE}  
  | {BEGIN | END} BACKUP  
  | SET DEFAULT CHARACTER SET charset  
  | SET LINGER TO linger_duration  
  | DROP LINGER  
  | {ENCRYPT WITH plugin_name [KEY key_name] | DECRYPT}

<add_sec_clause> ::= ADD <sec_file> [<sec_file> ...]
```
Multiple files can be added in one ADD clause:

```
ALTER DATABASE
  ADD FILE x LENGTH 8000
  FILE y LENGTH 8000
  FILE z
```

Multiple occurrences of `add_sec_clause` (ADD FILE clauses) are allowed; an ADD FILE clause that adds multiple files (as in the example above) can be mixed with others that add only one file. The statement was documented incorrectly in the old *InterBase 6 Language Reference*.

### Table 21. `ALTER DATABASE` Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_sec_clause</code></td>
<td>Adding a secondary database file</td>
</tr>
<tr>
<td><code>sec_file</code></td>
<td>File specification for secondary file</td>
</tr>
<tr>
<td><code>filepath</code></td>
<td>Full path and file name of the delta file or secondary database file</td>
</tr>
<tr>
<td><code>pagemum</code></td>
<td>Page number from which the secondary database file is to start</td>
</tr>
<tr>
<td><code>num</code></td>
<td>Maximum size of the secondary file in pages</td>
</tr>
<tr>
<td><code>diff_file</code></td>
<td>File path and name of the .delta file (difference file)</td>
</tr>
<tr>
<td><code>charset</code></td>
<td>New default character set of the database</td>
</tr>
<tr>
<td><code>linger_duration</code></td>
<td>Duration of linger delay in seconds; must be greater than or equal to 0 (zero)</td>
</tr>
<tr>
<td><code>plugin_name</code></td>
<td>The name of the encryption plugin</td>
</tr>
<tr>
<td><code>key_name</code></td>
<td>The name of the encryption key</td>
</tr>
</tbody>
</table>

The `ALTER DATABASE` statement can:

- add secondary files to a database
- switch a single-file database into and out of the “copy-safe” mode (DSQL only)
- set or unset the path and name of the delta file for physical backups (DSQL only)

`SCHEMA` is currently a synonym for `DATABASE`; this may change in a future version, so we recommend to always use `DATABASE`
**Who Can Alter the Database**

The `ALTER DATABASE` statement can be executed by:

- **Administrators**
- **Users with the `ALTER DATABASE` privilege**

**Parameters for `ALTER DATABASE`**

**ADD (FILE)**

Adds secondary files to the database. It is necessary to specify the full path to the file and the name of the secondary file. The description for the secondary file is similar to the one given for the `CREATE DATABASE` statement.

**ADD DIFFERENCE FILE**

Specifies the path and name of the delta file that stores any mutations to the database whenever it is switched to the “copy-safe” mode. This clause does not actually add any file. It just overrides the default name and path of the .delta file. To change the existing settings, you should delete the previously specified description of the .delta file using the `DROP DIFFERENCE FILE` clause before specifying the new description of the delta file. If the path and name of the .delta file are not overridden, the file will have the same path and name as the database, but with the .delta file extension.

If only a file name is specified, the .delta file will be created in the current directory of the server. On Windows, this will be the system directory—a very unwise location to store volatile user files and contrary to Windows file system rules.

**DROP DIFFERENCE FILE**

Deletes the description (path and name) of the .delta file specified previously in the `ADD DIFFERENCE FILE` clause. The file is not actually deleted. `DROP DIFFERENCE FILE` deletes the path and name of the .delta file from the database header. Next time the database is switched to the “copy-safe” mode, the default values will be used (i.e. the same path and name as those of the database, but with the .delta extension).

**BEGIN BACKUP**

Switches the database to the “copy-safe” mode. `ALTER DATABASE` with this clause freezes the main database file, making it possible to back it up safely using file system tools, even if users are connected and performing operations with data. Until the backup state of the database is reverted to `NORMAL`, all changes made to the database will be written to the .delta (difference) file.

Despite its syntax, a statement with the `BEGIN BACKUP` clause does not start a backup process but just creates the conditions for doing a task that requires the database file to be read-only temporarily.

**END BACKUP**

Switches the database from the “copy-safe” mode to the normal mode. A statement with this
clause merges the .delta file with the main database file and restores the normal operation of the database. Once the END BACKUP process starts, the conditions no longer exist for creating safe backups by means of file system tools.

Use of BEGIN BACKUP and END BACKUP and copying the database files with filesystem tools, is not safe with multi-file databases! Use this method only on single-file databases.

Making a safe backup with the gbak utility remains possible at all times, although it is not recommended running gbak while the database is in LOCKED or MERGE state.

SET DEFAULT CHARACTER SET
Changes the default character set of the database. This change does not affect existing data or columns. The new default character set will only be used in subsequent DDL commands.

SET LINGER TO
Sets the linger-delay. The linger-delay applies only to Firebird SuperServer, and is the number of seconds the server keeps a database file (and its caches) open after the last connection to that database was closed. This can help to improve performance at low cost, when the database is opened and closed frequently, by keeping resources “warm” for the next connection.

This mode can be useful for web applications - without a connection pool - where the connection to the database usually “lives” for a very short time.

The SET LINGER TO and DROP LINGER clauses can be combined in a single statement, but the last clause “wins”. For example, ALTER DATABASE SET LINGER TO 5 DROP LINGER will set the linger-delay to 0 (no linger), while ALTER DATABASE DROP LINGER SET LINGER to 5 will set the linger-delay to 5 seconds.

DROP LINGER
Drops the linger-delay (sets it to zero). Using DROP LINGER is equivalent to using SET LINGER TO 0.

Dropping LINGER is not an ideal solution for the occasional need to turn it off for some once-only condition where the server needs a forced shutdown. The gfix utility now has the -NoLinger switch, which will close the specified database immediately after the last attachment is gone, regardless of the LINGER setting in the database. The LINGER setting is retained and works normally the next time.

The same one-off override is also available through the Services API, using the tag isc_spb_prp_nolinger, e.g. (in one line):

```
fbsvcmgr host:service_mgr user sysdba password xxx
  action_properties dbname employee prp_nolinger
```

The DROP LINGER and SET LINGER TO clauses can be combined in a single
statement, but the last clause “wins”.

ENCRIPT WITH
See Encrypting a Database in the Security chapter.

DECRYPT
See Decrypting a Database in the Security chapter.

Examples of ALTER DATABASE Usage

1. Adding a secondary file to the database. As soon as 30000 pages are filled in the previous primary or secondary file, the Firebird engine will start adding data to the secondary file test4.fdb.

   ```sql
   ALTER DATABASE
   ADD FILE 'D:\test4.fdb'
   STARTING AT PAGE 30001;
   ```

2. Specifying the path and name of the delta file:

   ```sql
   ALTER DATABASE
   ADD DIFFERENCE FILE 'D:\test.diff';
   ```

3. Deleting the description of the delta file:

   ```sql
   ALTER DATABASE
   DROP DIFFERENCE FILE;
   ```

4. Switching the database to the “copy-safe” mode:

   ```sql
   ALTER DATABASE
   BEGIN BACKUP;
   ```

5. Switching the database back from the “copy-safe” mode to the normal operation mode:

   ```sql
   ALTER DATABASE
   END BACKUP;
   ```

6. Changing the default character set for a database to WIN1251

   ```sql
   ALTER DATABASE
   SET DEFAULT CHARACTER SET WIN1252;
   ```
7. Setting a *linger*-delay of 30 seconds

```
ALTER DATABASE
SET LINGER TO 30;
```

8. Encrypting the database with a plugin called DbCrypt

```
ALTER DATABASE
ENCRYPT WITH DbCrypt;
```

9. Decrypting the database

```
ALTER DATABASE
DECRYPT;
```

See also

`CREATE DATABASE, DROP DATABASE`

### 5.1.3. DROP DATABASE

*Used for*

Deleting the database to which you are currently connected

*Available in*

DSQL, ESQL

*Syntax*

```
DROP DATABASE
```

The `DROP DATABASE` statement deletes the current database. Before deleting a database, you have to connect to it. The statement deletes the primary file, all secondary files and all *shadow files*.

Contrary to `CREATE DATABASE` and `ALTER DATABASE`, `DROP SCHEMA` is not a valid alias for `DROP DATABASE`. This is intentional.

**Who Can Drop a Database**

The `DROP DATABASE` statement can be executed by:

- Administrators
- Users with the `DROP DATABASE` privilege
Example of DROP DATABASE

Deleting the current database

DROP DATABASE;

See also
CREATE DATABASE, ALTER DATABASE

5.2. SHADOW

A shadow is an exact, page-by-page copy of a database. Once a shadow is created, all changes made in the database are immediately reflected in the shadow. If the primary database file becomes unavailable for some reason, the DBMS will switch to the shadow.

This section describes how to create and delete shadow files.

5.2.1. CREATE SHADOW

Used for
Creating a shadow for the current database

Available in
DSQL, ESQL

Syntax

CREATE SHADOW <sh_num> [{AUTO | MANUAL}] [CONDITIONAL]
  'filepath' [LENGTH [=] num [PAGE[S]]]
  [<secondary_file> ...]

<secondary_file> ::= 
  FILE 'filepath'
  [STARTING [AT [PAGE]] pagenum]
  [LENGTH [=] num [PAGE[S]]]

Table 22. CREATE SHADOW Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh_num</td>
<td>Shadow number — a positive number identifying the shadow set</td>
</tr>
<tr>
<td>filepath</td>
<td>The name of the shadow file and the path to it, in accord with the rules of the operating system</td>
</tr>
<tr>
<td>num</td>
<td>Maximum shadow size, in pages</td>
</tr>
<tr>
<td>secondary_file</td>
<td>Secondary file specification</td>
</tr>
<tr>
<td>page_num</td>
<td>The number of the page at which the secondary shadow file should start</td>
</tr>
</tbody>
</table>

The CREATE SHADOW statement creates a new shadow. The shadow starts duplicating the database
right at the moment it is created. It is not possible for a user to connect to a shadow.

Like a database, a shadow may be multi-file. The number and size of a shadow's files are not related to the number and size of the files of database it is shadowing.

The page size for shadow files is set to be equal to the database page size and cannot be changed.

If a calamity occurs involving the original database, the system converts the shadow to a copy of the database and switches to it. The shadow is then unavailable. What happens next depends on the MODE option.

**AUTO | MANUAL Modes**

When a shadow is converted to a database, it becomes unavailable. A shadow might alternatively become unavailable because someone accidentally deletes its file, or the disk space where the shadow files are stored is exhausted or is itself damaged.

- If the AUTO mode is selected (the default value), shadowing ceases automatically, all references to it are deleted from the database header, and the database continues functioning normally.

  If the CONDITIONAL option was set, the system will attempt to create a new shadow to replace the lost one. It does not always succeed, however, and a new one may need to be created manually.

- If the MANUAL mode attribute is set when the shadow becomes unavailable, all attempts to connect to the database and to query it will produce error messages. The database will remain inaccessible until either the shadow again becomes available, or the database administrator deletes it using the DROP SHADOW statement. MANUAL should be selected if continuous shadowing is more important than uninterrupted operation of the database.

**Options for CREATE SHADOW**

LENGTH

Specifies the maximum size of the primary or secondary shadow file in pages. The LENGTH value does not affect the size of the only shadow file, nor the last if it is a set. The last (or only) file will keep automatically growing as long as it is necessary.

STARTING AT

Specifies the shadow page number at which the next shadow file should start. The system will start adding new data to the next shadow file when the previous file is filled with data up to the specified page number.

You can verify the sizes, names and location of the shadow files by connecting to the database using *isql* and running the command *SHOW DATABASE*;

**Who Can Create a Shadow**

The CREATE SHADOW statement can be executed by:

- Administrators
• Users with the ALTER DATABASE privilege

**Examples Using CREATE SHADOW**

1. Creating a shadow for the current database as “shadow number 1”:

   ```sql
   CREATE SHADOW 1 'g:\data\test.shd';
   ```

2. Creating a multi-file shadow for the current database as “shadow number 2”:

   ```sql
   CREATE SHADOW 2 'g:\data\test.sh1'
       LENGTH 8000 PAGES
       FILE 'g:\data\test.sh2';
   ```

See also

CREATE DATABASE, DROP SHADOW

**5.2.2. DROP SHADOW**

*Used for*
Deleting a shadow from the current database

*Available in*
DSQL, ESQL

*Syntax*

```sql
DROP SHADOW sh_num
   [{DELETE | PRESERVE} FILE]
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh_num</td>
<td>Shadow number — a positive number identifying the shadow set</td>
</tr>
</tbody>
</table>

The DROP SHADOW statement deletes the specified shadow for the current database. When a shadow is dropped, all files related to it are deleted and shadowing to the specified `sh_num` ceases. The optional `DELETE FILE` clause makes this behaviour explicit. On the contrary, the `PRESERVE FILE` clause will remove the shadow from the database, but the file itself will not be deleted.

**Who Can Drop a Shadow**

The DROP SHADOW statement can be executed by:

• **Administrators**

• Users with the ALTER DATABASE privilege
Example of DROP SHADOW

Deleting “shadow number 1”.

DROP SHADOW 1;

See also
CREATE SHADOW

5.3. DOMAIN

DOMAIN is one of the object types in a relational database. A domain is created as a specific data type with some attributes attached to it. Once it has been defined in the database, it can be reused repeatedly to define table columns, PSQL arguments and PSQL local variables. Those objects inherit all of the attributes of the domain. Some attributes can be overridden when the new object is defined, if required.

This section describes the syntax of statements used to create, modify and delete domains. A detailed description of domains and their usage can be found in Custom Data Types — Domains.

5.3.1. CREATE DOMAIN

Used for
Creating a new domain

Available in
DSQL, ESQL

Syntax

CREATE DOMAIN name [AS] <datatype>
  [DEFAULT {<literal> | NULL | <context_var>}]  
  [NOT NULL] [CHECK (<dom_condition>)]
  [COLLATE collation_name]

<datatype> ::=<scalar_datatype> | <blob_datatype> | <array_datatype>

<scalar_datatype> ::= <!-- See Scalar Data Types Syntax -->

<blob_datatype> ::=<!-- See BLOB Data Types Syntax -->

<array_datatype> ::= <!-- See Array Data Types Syntax -->

<dom_condition> ::=<val> <operator> <val>
| <val> [NOT] BETWEEN <val> AND <val> |
| <val> [NOT] IN ({{<val> [, , <val> ...]} | <select_list>}) |
| <val> IS [NOT] NULL |
| <val> IS [NOT] DISTINCT FROM <val> |
| <val> [NOT] CONTAINING <val> |
| <val> [NOT] STARTING [WITH] <val> |
| <val> [NOT] LIKE <val> [ESCAPE <val>] |
| <val> [NOT] SIMILAR TO <val> [ESCAPE <val>] |
| <val> <operator> {ALL | SOME | ANY} (<select_list>) |
| [NOT] EXISTS (<select_expr>) |
| [NOT] SINGULAR (<select_expr>) |
| (<dom_condition>) |
| NOT <dom_condition> |
| <dom_condition> OR <dom_condition> |
| <dom_condition> AND <dom_condition> |

<operator> ::= |
  <> | != | ^= | ~= | = | < | > | <= | >= |
  | !< | ^< | ~< | !> | ^> | ~> |

<val> ::= |
  VALUE |
  | <literal> |
  | <context_var> |
  | <expression> |
  | NULL |
  | NEXT VALUE FOR genname |
  | GEN_ID(genname, <val>) |
  | CAST(<val> AS <cast_type>) |
  | (<select_one>) |
  | func([[<val> [, , <val> ...]]]) |

<cast_type> ::= <domain_or_non_array_type> | <array_datatype> |

<domain_or_non_array_type> ::= |
  !! See Scalar Data Types Syntax !!
The `CREATE DOMAIN` statement creates a new domain.

Any SQL data type can be specified as the domain type.

**Type-specific Details**

**Array Types**

- If the domain is to be an array, the base type can be any SQL data type except `BLOB` and `array`.
- The dimensions of the array are specified between square brackets. (In the Syntax block, these brackets appear in quotes to distinguish them from the square brackets that identify optional syntax elements.)
- For each array dimension, one or two integer numbers define the lower and upper boundaries of its index range:
  - By default, arrays are 1-based. The lower boundary is implicit and only the upper boundary need be specified. A single number smaller than 1 defines the range `num..1` and a number greater than 1 defines the range `1..num`.
  - Two numbers separated by a colon (`:`) and optional whitespace, the second greater than the first, can be used to define the range explicitly. One or both boundaries can be less than zero, as long as the upper boundary is greater than the lower.
- When the array has multiple dimensions, the range definitions for each dimension must be separated by commas and optional whitespace.
- Subscripts are validated *only* if an array actually exists. It means that no error messages regarding invalid subscripts will be returned if selecting a specific element returns nothing or if an array field is `NULL`.

**String Types**

You can use the `CHARACTER SET` clause to specify the character set for the `CHAR`, `VARCHAR` and `BLOB` (`SUB_TYPE TEXT`) types. If the character set is not specified, the character set specified as `DEFAULT CHARACTER SET` of the database will be used. If no character set was specified then, the character set `NONE` is applied by default when you create a character domain.

With character set `NONE`, character data are stored and retrieved the way they were submitted. Data in any encoding can be added to a column based on such
a domain, but it is impossible to add this data to a column with a different encoding. Because no transliteration is performed between the source and destination encodings, errors may result.

**DEFAULT Clause**

The optional `DEFAULT` clause allows you to specify a default value for the domain. This value will be added to the table column that inherits this domain when the `INSERT` statement is executed, if no value is specified for it in the DML statement. Local variables and arguments in PSQL modules that reference this domain will be initialized with the default value. For the default value, use a literal of a compatible type or a context variable of a compatible type.

**NOT NULL Constraint**

Columns and variables based on a domain with the `NOT NULL` constraint will be prevented from being written as `NULL`, i.e., a value is *required*.

When creating a domain, take care to avoid specifying limitations that would contradict one another. For instance, `NOT NULL` and `DEFAULT NULL` are contradictory.

**CHECK Constraint(s)**

The optional `CHECK` clause specifies constraints for the domain. A domain constraint specifies conditions that must be satisfied by the values of table columns or variables that inherit from the domain. A condition must be enclosed in parentheses. A condition is a logical expression (also called a predicate) that can return the Boolean results `TRUE`, `FALSE` and `UNKNOWN`. A condition is considered satisfied if the predicate returns the value `TRUE` or “unknown value” (equivalent to `NULL`). If the predicate returns `FALSE`, the condition for acceptance is not met.

**VALUE Keyword**

The keyword `VALUE` in a domain constraint substitutes for the table column that is based on this domain or for a variable in a PSQL module. It contains the value assigned to the variable or the table column. `VALUE` can be used anywhere in the `CHECK` constraint, though it is usually used in the left part of the condition.

**COLLATE**

The optional `COLLATE` clause allows you to specify the collation sequence if the domain is based on one of the string data types, including `BLOB`s with text subtypes. If no collation sequence is specified, the collation sequence will be the one that is default for the specified character set at the time the domain is created.

**Who Can Create a Domain**

The `CREATE DOMAIN` statement can be executed by:

- **Administrators**
- Users with the `CREATE DOMAIN` privilege
CREATE DOMAIN Examples

1. Creating a domain that can take values greater than 1,000, with a default value of 10,000.

```sql
CREATE DOMAIN CUSTNO AS
    INTEGER DEFAULT 10000
    CHECK (VALUE > 1000);
```

2. Creating a domain that can take the values 'Yes' and 'No' in the default character set specified during the creation of the database.

```sql
CREATE DOMAIN D_BOOLEAN AS
    CHAR(3) CHECK (VALUE IN ('Yes', 'No'));
```

3. Creating a domain with the UTF8 character set and the UNICODE_CI_AI collation sequence.

```sql
CREATE DOMAIN FIRSTNAME AS
    VARCHAR(30) CHARACTER SET UTF8
    COLLATE UNICODE_CI_AI;
```

4. Creating a domain of the DATE type that will not accept NULL and uses the current date as the default value.

```sql
CREATE DOMAIN D_DATE AS
    DATE DEFAULT CURRENT_DATE
    NOT NULL;
```

5. Creating a domain defined as an array of 2 elements of the NUMERIC(18, 3) type. The starting array index is 1.

```sql
CREATE DOMAIN D_POINT AS
    NUMERIC(18, 3) [2];
```

- Domains defined over an array type may be used only to define table columns. You cannot use array domains to define local variables in PSQL modules.

6. Creating a domain whose elements can be only country codes defined in the COUNTRY table.

```sql
CREATE DOMAIN D_COUNTRYCODE AS CHAR(3)
    CHECK (EXISTS(SELECT * FROM COUNTRY
        WHERE COUNTRYCODE = VALUE));
```

- The example is given only to show the possibility of using predicates with
queries in the domain test condition. It is not recommended to create this style of domain in practice unless the lookup table contains data that are never deleted.

See also
ALTER DOMAIN, DROP DOMAIN

5.3.2. ALTER DOMAIN

Used for
Altering the current attributes of a domain or renaming it

Available in
DSQL, ESQL

Syntax

```
ALTER DOMAIN domain_name
    [TO new_name]
    [TYPE <datatype>]
    [{SET DEFAULT {<literal> | NULL | <context_var>} | DROP DEFAULT}]
    [{SET | DROP} NOT NULL]
    [{ADD [CONSTRAINT] CHECK (<dom_condition>) | DROP CONSTRAINT}]
```

\[<datatype> ::= \]
\[<scalar_datatype> | <blob_datatype>\]

\[<scalar_datatype> ::= \]
\[!! See Scalar Data Types Syntax !!\]

\[<blob_datatype> ::= \]
\[!! See BLOB Data Types Syntax !!\]

\[!! See also CREATE DOMAIN Syntax !!\]

Table 25. ALTER DOMAIN Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_name</td>
<td>New name for domain, consisting of up to 31 characters</td>
</tr>
<tr>
<td>literal</td>
<td>A literal value that is compatible with \textit{datatype}</td>
</tr>
<tr>
<td>context_var</td>
<td>Any context variable whose type is compatible with \textit{datatype}</td>
</tr>
</tbody>
</table>

The \textit{ALTER DOMAIN} statement enables changes to the current attributes of a domain, including its name. You can make any number of domain alterations in one \textit{ALTER DOMAIN} statement.

\textbf{ALTER DOMAIN clauses}
TO name

Use the TO clause to rename the domain, as long as there are no dependencies on the domain, i.e. table columns, local variables or procedure arguments referencing it.

SET DEFAULT

With the SET DEFAULT clause you can set a new default value. If the domain already has a default value, there is no need to delete it first — it will be replaced by the new one.

DROP DEFAULT

Use this clause to delete a previously specified default value and replace it with NULL.

SET NOT NULL

Use this class to add a NOT NULL constraint to the domain; columns or parameters of this domain will be prevented from being written as NULL, i.e., a value is required.

Adding a NOT NULL constraint to an existing domain will subject all columns using this comain to a full data validation, so ensure that the columns have no nulls before attempting the change.

DROP NOT NULL

Drop the NOT NULL constraint from the domain.

An explicit NOT NULL constraint on a column that depends on a domain prevails over the domain. In this situation, the modification of the domain to make it nullable does not propagate to the column.

ADD CONSTRAINT CHECK

Use the ADD CONSTRAINT CHECK clause to add a CHECK constraint to the domain. If the domain already has a CHECK constraint, it will have to be deleted first, using an ALTER DOMAIN statement that includes a DROP CONSTRAINT clause.

TYPE

The TYPE clause is used to change the data type of the domain to a different, compatible one. The system will forbid any change to the type that could result in data loss. An example would be if the number of characters in the new type were smaller than in the existing type.

When you alter the attributes of a domain, existing PSQL code may become invalid. For information on how to detect it, read the piece entitled The RDB$VALID_BLR Field in Appendix A.

What ALTER DOMAIN Cannot Alter

• If the domain was declared as an array, it is not possible to change its type or its dimensions; nor can any other type be changed to an array type.

• There is no way to change the default collation without dropping the domain and recreating it with the desired attributes.
**Who Can Alter a Domain**

The `ALTER DOMAIN` statement can be executed by:

- **Administrators**
- The owner of the domain
- Users with the `ALTER ANY DOMAIN` privilege

Domain alterations can be prevented by dependencies from objects to which the user does not have sufficient privileges.

**ALTER DOMAIN Examples**

1. Changing the data type to `INTEGER` and setting or changing the default value to 2,000:

   ```sql
   ALTER DOMAIN CUSTNO
   TYPE INTEGER
   SET DEFAULT 2000;
   ```

2. Renaming a domain.

   ```sql
   ALTER DOMAIN D_BOOLEAN TO D_BOOL;
   ```

3. Deleting the default value and adding a constraint for the domain:

   ```sql
   ALTER DOMAIN D_DATE
   DROP DEFAULT
   ADD CONSTRAINT CHECK (VALUE >= date '01.01.2000');
   ```

4. Changing the `CHECK` constraint:

   ```sql
   ALTER DOMAIN D_DATE
   DROP CONSTRAINT;
   ALTER DOMAIN D_DATE
   ADD CONSTRAINT CHECK
   (VALUE BETWEEN date '01.01.1900' AND date '31.12.2100');
   ```

5. Changing the data type to increase the permitted number of characters:

   ```sql
   ALTER DOMAIN FIRSTNAME
   TYPE VARCHAR(50) CHARACTER SET UTF8;
   ```

6. Adding a `NOT NULL` constraint:
ALTER DOMAIN FIRSTNAME
    SET NOT NULL;

7. Removing a NOT NULL constraint:

ALTER DOMAIN FIRSTNAME
    DROP NOT NULL;

See also
CREATE DOMAIN, DROP DOMAIN

5.3.3. DROP DOMAIN

Used for
Deleting an existing domain

Available in
DSQL, ESQL

Syntax

    DROP DOMAIN domain_name

The DROP DOMAIN statement deletes a domain that exists in the database. It is not possible to delete a domain if it is referenced by any database table columns or used in any PSQL module. In order to delete a domain that is in use, all columns in all tables that refer to the domain will have to be dropped and all references to the domain will have to be removed from PSQL modules.

Who Can Drop a Domain

The DROP DOMAIN statement can be executed by:

• Administrators
• The owner of the domain
• Users with the DROP ANY DOMAIN privilege

Example of DROP DOMAIN

Deleting the COUNTRYNAME domain

    DROP DOMAIN COUNTRYNAME;

See also
CREATE DOMAIN, ALTER DOMAIN
5.4. TABLE

As a relational DBMS, Firebird stores data in tables. A table is a flat, two-dimensional structure containing any number of rows. Table rows are often called *records*.

All rows in a table have the same structure and consist of columns. Table columns are often called *fields*. A table must have at least one column. Each column contains a single type of SQL data.

This section describes how to create, alter and delete tables in a database.

5.4.1. CREATE TABLE

*Used for*
creating a new table (relation)

*Available in*
DSQL, ESQL

*Syntax*

```
CREATE [GLOBAL TEMPORARY] TABLE tablename
[EXTERNAL [FILE] 'filenamespec']
(coll_def) [, {<col_def> | <tconstraint>} ...])
[ON COMMIT {DELETE | PRESERVE} ROWS]

<col_def> ::= 
  <regular_col_def>
  | <computed_col_def>
  | <identity_col_def>

<regular_col_def> ::= 
  colname {<datatype> | domainname}
  [DEFAULT {<literal> | NULL | <context_var}>]
  [<col_constraint> ...]
  [COLLATE collation_name]

<computed_col_def> ::= 
  colname [{<datatype> | domainname}]
  {COMPUTED [BY] | GENERATED ALWAYS AS} (<expression>)

<identity_col_def> ::= 
  colname {<datatype> | domainname}
  GENERATED BY DEFAULT AS IDENTITY [(START WITH startvalue)]
  [<col_constraint> ...]

<datatype> ::= 
  <scalar_datatype> | <blob_datatype> | <array_datatype>
```

!! See Scalar Data Types Syntax !!
<blob_datatype> ::= 
!! See BLOB Data Types Syntax !!

<array_datatype> ::= 
!! See Array Data Types Syntax !!

<col_constraint> ::= 
[CONSTRAINT constr_name] 
{ PRIMARY KEY [using_index] 
| UNIQUE [using_index] 
| REFERENCES other_table [(colname)] [using_index] 
   [ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}] 
   [ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}] 
| CHECK (check_condition) 
| NOT NULL } 

<tconstraint> ::= 
[CONSTRAINT constr_name] 
{ PRIMARY KEY (col_list) [using_index] 
| UNIQUE (col_list) [using_index] 
| FOREIGN KEY (col_list) 
   REFERENCES other_table [(col_list)] [using_index] 
   [ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}] 
   [ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}] 
| CHECK (check_condition) } 

<col_list> ::= colname [, colname ...] 

<using_index> ::= USING 
   [ASC[ENDING] | DESC[ENDING]] INDEX indexname 

<check_condition> ::= 
<val> <operator> <val> 
| <val> [NOT] BETWEEN <val> AND <val> 
| <val> [NOT] IN (val [, val ...] | <select_list>) 
| <val> IS [NOT] NULL 
| <val> IS [NOT] DISTINCT FROM <val> 
| <val> [NOT] CONTAINING <val> 
| <val> [NOT] STARTING [WITH] <val> 
| <val> [NOT] LIKE <val> [ESCAPE <val>] 
| <val> [NOT] SIMILAR TO <val> [ESCAPE <val>] 
| <val> <operator> {ALL | SOME | ANY} (<select_list>) 
| [NOT] EXISTS (<select_expr>) 
| [NOT] SINGULAR (<select_expr>) 
| (check_condition) 
| NOT <check_condition> 
| <check_condition> OR <check_condition> 
| <check_condition> AND <check_condition> 

<operator> ::=
Table 26. CREATE TABLE Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tablename</td>
<td>Name (identifier) for the table. It may consist of up to 31 characters and must be unique in the database.</td>
</tr>
<tr>
<td>filespec</td>
<td>File specification (only for external tables). Full file name and path, enclosed in single quotes, correct for the local file system and located on a storage device that is physically connected to Firebird’s host computer.</td>
</tr>
<tr>
<td>colname</td>
<td>Name (identifier) for a column in the table. May consist of up to 31 characters and must be unique in the table.</td>
</tr>
<tr>
<td>datatype</td>
<td>SQL data type</td>
</tr>
<tr>
<td>domain_name</td>
<td>Domain name</td>
</tr>
<tr>
<td>start_value</td>
<td>The initial value of the identity column</td>
</tr>
<tr>
<td>col_constraint</td>
<td>Column constraint</td>
</tr>
<tr>
<td>tconstraint</td>
<td>Table constraint</td>
</tr>
<tr>
<td>constr_name</td>
<td>The name (identifier) of a constraint. May consist of up to 31 characters.</td>
</tr>
<tr>
<td>other_table</td>
<td>The name of the table referenced by the foreign key constraint</td>
</tr>
<tr>
<td>other_col</td>
<td>The name of the column in other_table that is referenced by the foreign key</td>
</tr>
<tr>
<td>literal</td>
<td>A literal value that is allowed in the given context</td>
</tr>
<tr>
<td>context_var</td>
<td>Any context variable whose data type is allowed in the given context</td>
</tr>
<tr>
<td>check_condition</td>
<td>The condition applied to a CHECK constraint, that will resolve as either true, false or NULL</td>
</tr>
<tr>
<td><strong>Parameter</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
</tr>
<tr>
<td>collation</td>
<td>Collation</td>
</tr>
<tr>
<td>select_one</td>
<td>A scalar SELECT statement — selecting one column and returning only one row</td>
</tr>
<tr>
<td>select_list</td>
<td>A SELECT statement selecting one column and returning zero or more rows</td>
</tr>
<tr>
<td>select_expr</td>
<td>A SELECT statement selecting one or more columns and returning zero or more rows</td>
</tr>
<tr>
<td>expression</td>
<td>An expression resolving to a value that is allowed in the given context</td>
</tr>
<tr>
<td>genname</td>
<td>Sequence (generator) name</td>
</tr>
<tr>
<td>func</td>
<td>Internal function or UDF</td>
</tr>
</tbody>
</table>

The `CREATE TABLE` statement creates a new table. Any user can create it and its name must be unique among the names of all tables, views and stored procedures in the database.

A table must contain at least one column that is not computed, and the names of columns must be unique in the table.

A column must have either an explicit SQL data type, the name of a domain whose attributes will be copied for the column, or be defined as `COMPUTED BY` an expression (a calculated field).

A table may have any number of table constraints, including none.

**Character Columns**

You can use the `CHARACTER SET` clause to specify the character set for the `CHAR`, `VARCHAR` and `BLOB` (text subtype) types. If the character set is not specified, the default character set of the database - at time of the creation of the column - will be used. If the database has no default character set, the `NONE` character set is applied. In this case, data is stored and retrieved the way it was submitted. Data in any encoding can be added to such a column, but it is not possible to add this data to a column with a different encoding. No transliteration is performed between the source and destination encodings, which may result in errors.

The optional `COLLATE` clause allows you to specify the collation sequence for character data types, including `BLOB SUB_TYPE TEXT`. If no collation sequence is specified, the default collation sequence for the specified character set - at time of the creation of the column - is applied.

**Setting a DEFAULT Value**

The optional `DEFAULT` clause allows you to specify the default value for the table column. This value will be added to the column when an `INSERT` statement is executed if no value was specified for it and that column was omitted from the `INSERT` command.

The default value can be a literal of a compatible type, a context variable that is type-compatible with the data type of the column, or `NULL`, if the column allows it. If no default value is explicitly specified, `NULL` is implied.
An expression cannot be used as a default value.

**Domain-based Columns**

To define a column, you can use a previously defined domain. If the definition of a column is based on a domain, it may contain a new default value, additional CHECK constraints, and a COLLATE clause that will override the values specified in the domain definition. The definition of such a column may contain additional column constraints (for instance, NOT NULL), if the domain does not have it.

It is not possible to define a domain-based column that is nullable if the domain was defined with the NOT NULL attribute. If you want to have a domain that might be used for defining both nullable and non-nullable columns and variables, it is better practice defining the domain nullable and apply NOT NULL in the downstream column definitions and variable declarations.

**Identity Columns (autoincrement)**

Identity columns can be defined using the `GENERATED BY DEFAULT AS IDENTITY` clause. The identity column is the column associated with internal sequence generator. Its value is set automatically every time it is not specified in the `INSERT` statement. The optional `START WITH` clause allows you to specify an initial value other than 1.

**Incorrect `START WITH` behaviour**

The SQL standard requires that `START WITH` specifies the first value to be generated. Unfortunately, the current implementation in Firebird instead uses the specified value as the initial value of the internal generator backing the identity column. That means that right now it specifies the value before the first value that is generated.

This will be fixed in Firebird 4, see also [CORE-6376](#).

**Rules**

- The data type of an identity column must be an exact number type with zero scale. Allowed types are thus SMALLINT, INTEGER, BIGINT, NUMERIC(p[,0]) and DECIMAL(p[,0]).

- An identity column cannot have a DEFAULT or COMPUTED value.

  - An identity column cannot be altered to become a regular column. The reverse is also true. Firebird 4 will introduce the option to alter an identity column to a regular column.

  - Identity columns are implicitly NOT NULL (non-nullable).

  - Uniqueness is not enforced automatically. A UNIQUE or PRIMARY KEY constraint is required to guarantee uniqueness.

  - The use of other methods of generating key values for identity columns, e.g. by trigger-generator code or by allowing users to change or add them, is discouraged to avoid unexpected key violations.
Calculated Fields

Calculated fields can be defined with the `COMPUTED [BY]` or `GENERATED ALWAYS AS` clause (according to the SQL:2003 standard). They mean the same. Describing the data type is not required (but possible) for calculated fields, as the DBMS calculates and stores the appropriate type as a result of the expression analysis. Appropriate operations for the data types included in an expression must be specified precisely.

If the data type is explicitly specified for a calculated field, the calculation result is converted to the specified type. This means, for instance, that the result of a numeric expression could be rendered as a string.

In a query that selects a `COMPUTED BY` column, the expression is evaluated for each row of the selected data.

Instead of a computed column, in some cases it makes sense to use a regular column whose value is evaluated in triggers for adding and updating data. It may reduce the performance of inserting/updating records, but it will increase the performance of data selection.

Defining an Array Column

- If the column is to be an array, the base type can be any SQL data type except `BLOB` and array.
- The dimensions of the array are specified between square brackets. (In the Syntax block these brackets appear in quotes to distinguish them from the square brackets that identify optional syntax elements.)
- For each array dimension, one or two integer numbers define the lower and upper boundaries of its index range:
  - By default, arrays are 1-based. The lower boundary is implicit and only the upper boundary need be specified. A single number smaller than 1 defines the range `num..1` and a number greater than 1 defines the range `1..num`.
  - Two numbers separated by a colon (`:`) and optional whitespace, the second greater than the first, can be used to define the range explicitly. One or both boundaries can be less than zero, as long as the upper boundary is greater than the lower.
- When the array has multiple dimensions, the range definitions for each dimension must be separated by commas and optional whitespace.
- Subscripts are validated only if an array actually exists. It means that no error messages regarding invalid subscripts will be returned if selecting a specific element returns nothing or if an array field is `NULL`.

Constraints

Five types of constraints can be specified. They are:

- Primary key (` PRIMARY KEY `)
- Unique key (`UNIQUE`)
• Foreign key (REFERENCES)
• CHECK constraint (CHECK)
• NOT NULL constraint (NOT NULL)

Constraints can be specified at column level (“column constraints”) or at table level (“table constraints”). Table-level constraints are required when keys (unique constraint, Primary Key, Foreign Key) consist of multiple columns and when a CHECK constraint involves other columns in the row besides the column being defined. The NOT NULL constraint can only be specified as a column constraint. Syntax for some types of constraint may differ slightly according to whether the constraint is defined at the column or table level.

• A column-level constraint is specified during a column definition, after all column attributes except COLLATION are specified, and can involve only the column specified in that definition
• A table-level constraints can only be specified after the definitions of the columns used in the constraint.
• Table-level constraints are a more flexible way to set constraints, since they can cater for constraints involving multiple columns
• You can mix column-level and table-level constraints in the same CREATE TABLE statement

The system automatically creates the corresponding index for a primary key (PRIMARY KEY), a unique key (UNIQUE) and a foreign key (REFERENCES for a column-level constraint, FOREIGN KEY REFERENCES for one at the table level).

Names for Constraints and Their Indexes

Column-level constraints and their indexes are named automatically:

• The constraint name has the form INTEG_n, where n represents one or more digits
• The index name has the form RDB$PRIMARYn (for a primary key index), RDB$FOREIGNn (for a foreign key index) or RDB$n (for a unique key index). Again, n represents one or more digits.

Automatic naming of table-level constraints and their indexes follows the same pattern, unless the names are supplied explicitly.

Named Constraints

A constraint can be named explicitly if the CONSTRAINT clause is used for its definition. While the CONSTRAINT clause is optional for defining column-level constraints, it is mandatory for table-level constraints. By default, the constraint index will have the same name as the constraint. If a different name is wanted for the constraint index, a USING clause can be included.

The USING Clause

The USING clause allows you to specify a user-defined name for the index that is created automatically and, optionally, to define the direction of the index — either ascending (the default) or descending.
The **PRIMARY KEY** constraint is built on one or more *key columns*, where each column has the **NOT NULL** constraint specified. The values across the key columns in any row must be unique. A table can have only one primary key.

- A single-column Primary Key can be defined as a column level or a table-level constraint
- A multi-column Primary Key must be specified as a table-level constraint

**The UNIQUE Constraint**

The **UNIQUE** constraint defines the requirement of content uniqueness for the values in a key throughout the table. A table can contain any number of unique key constraints.

As with the Primary Key, the Unique constraint can be multi-column. If so, it must be specified as a table-level constraint.

**NULL in Unique Keys**

Firebird’s SQL-99-compliant rules for **UNIQUE** constraints allow one or more **NULLs** in a column with a **UNIQUE** constraint. That makes it possible to define a **UNIQUE** constraint on a column that does not have the **NOT NULL** constraint.

For **UNIQUE** keys that span multiple columns, the logic is a little complicated:

- Multiple rows having null in all the columns of the key are allowed
- Multiple rows having keys with different combinations of nulls and non-null values are allowed
- Multiple rows having the same key columns null and the rest filled with non-null values are allowed, provided the values differ in at least one column
- Multiple rows having the same key columns null and the rest filled with non-null values that are the same in every column will violate the constraint

The rules for uniqueness can be summarised thus:

> In principle, all nulls are considered distinct. However, if two rows have exactly the same key columns filled with non-null values, the **NULL** columns are ignored and the uniqueness is determined on the non-null columns as though they constituted the entire key.

**Illustration**

```
RECREATE TABLE t( x int, y int, z int, unique(x,y,z));
INSERT INTO t values( NULL, 1, 1 );
INSERT INTO t values( NULL, NULL, 1 );
INSERT INTO t values( NULL, NULL, NULL );
INSERT INTO t values( NULL, NULL, NULL ); -- Permitted
INSERT INTO t values( NULL, NULL, 1 ); -- Not permitted
```
A Foreign Key ensures that the participating column(s) can contain only values that also exist in the referenced column(s) in the master table. These referenced columns are often called target columns. They must be the primary key or a unique key in the target table. They need not have a NOT NULL constraint defined on them although, if they are the primary key, they will, of course, have that constraint.

The foreign key columns in the referencing table itself do not require a NOT NULL constraint.

A single-column Foreign Key can be defined in the column declaration, using the keyword REFERENCES:

```sql
...,
  ARTIFACT_ID INTEGER REFERENCES COLLECTION (ARTIFACT_ID),
```

The column ARTIFACT_ID in the example references a column of the same name in the table COLLECTIONS.

Both single-column and multi-column foreign keys can be defined at the table level. For a multi-column Foreign Key, the table-level declaration is the only option. This method also enables the provision of an optional name for the constraint:

```sql
...,
  CONSTRAINT FK_ARTSOURCE FOREIGN KEY(DEALER_ID, COUNTRY)
    REFERENCES DEALER (DEALER_ID, COUNTRY),
```

Notice that the column names in the referenced (“master”) table may differ from those in the Foreign Key.

If no target columns are specified, the Foreign Key automatically references the target table's Primary Key.

### Foreign Key Actions

With the sub-clauses ON UPDATE and ON DELETE it is possible to specify an action to be taken on the affected foreign key column(s) when referenced values in the master table are changed:

- **NO ACTION**
  
  (the default) - Nothing is done

- **CASCADE**

  The change in the master table is propagated to the corresponding row(s) in the child table. If a key value changes, the corresponding key in the child records changes to the new value; if the master row is deleted, the child records are deleted.

- **SET DEFAULT**

  The Foreign Key columns in the affected rows will be set to their default values as they were
when the foreign key constraint was defined.

**SET NULL**

The Foreign Key columns in the affected rows will be set to **NULL**.

The specified action, or the default **NO ACTION**, could cause a Foreign Key column to become invalid. For example, it could get a value that is not present in the master table, or it could become **NULL** while the column has a **NOT NULL** constraint. Such conditions will cause the operation on the master table to fail with an error message.

**Example**

```sql
...
 CONSTRAINT FK_ORDERS_CUST
 FOREIGN KEY (CUSTOMER) REFERENCES CUSTOMERS (ID)
 ON UPDATE CASCADE ON DELETE SET NULL
```

**CHECK Constraint**

The **CHECK** constraint defines the condition the values inserted in this column must satisfy. A condition is a logical expression (also called a predicate) that can return the **TRUE**, **FALSE** and **UNKNOWN** values. A condition is considered satisfied if the predicate returns **TRUE** or value **UNKNOWN** (equivalent to **NULL**). If the predicate returns **FALSE**, the value will not be accepted. This condition is used for inserting a new row into the table (the **INSERT** statement) and for updating the existing value of the table column (the **UPDATE** statement) and also for statements where one of these actions may take place (**UPDATE OR INSERT**, **MERGE**).

A **CHECK** constraint on a domain-based column does not replace an existing **CHECK** condition on the domain, but becomes an addition to it. The Firebird engine has no way, during definition, to verify that the extra **CHECK** does not conflict with the existing one.

**Example**

with two column-level constraints and one at table-level:

```sql
CREATE TABLE PLACES (  
...
 LAT DECIMAL(9, 6) CHECK (ABS(LAT) <= 90),
 LON DECIMAL(9, 6) CHECK (ABS(LON) <= 180),
...
 CONSTRAINT CHK_POLES CHECK (ABS(LAT) < 90 OR LON = 0)
);
```
**NOT NULL constraint**

In Firebird, columns are nullable by default. The **NOT NULL** constraint specifies that the column cannot take **NULL** in place of a value.

A **NOT NULL** constraint can only be defined as a column constraint, not as a table constraint.

**Who Can Create a Table**

The **CREATE TABLE** statement can be executed by:

- **Administrators**
- Users with the **CREATE TABLE** privilege

The user executing the **CREATE TABLE** statement becomes the owner of the table.

**CREATE TABLE Examples**

1. Creating the **COUNTRY** table with the primary key specified as a column constraint.

   ```sql
   CREATE TABLE COUNTRY (  
     COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,  
     CURRENCY VARCHAR(10) NOT NULL  
   );
   ```

2. Creating the **STOCK** table with the named primary key specified at the column level and the named unique key specified at the table level.

   ```sql
   CREATE TABLE STOCK (  
     MODEL     SMALLINT NOT NULL CONSTRAINT PK_STOCK PRIMARY KEY,  
     MODELNAME CHAR(10) NOT NULL,  
     ITEMID    INTEGER NOT NULL,  
     CONSTRAINT MOD_UNIQUE UNIQUE (MODELNAME, ITEMID)  
   );
   ```

3. Creating the **JOB** table with a primary key constraint spanning two columns, a foreign key constraint for the **COUNTRY** table and a table-level **CHECK** constraint. The table also contains an array of 5 elements.

   ```sql
   CREATE TABLE JOB (  
     JOB_CODE        JOBCODE NOT NULL,  
     JOB_GRADE       JOBGRADE NOT NULL,  
     JOB_COUNTRY     COUNTRYNAME,  
     JOB_TITLE       VARCHAR(25) NOT NULL,  
     MIN_SALARY      NUMERIC(18, 2) DEFAULT 0 NOT NULL,  
     MAX_SALARY      NUMERIC(18, 2) NOT NULL,  
     JOB_REQUIREMENT BLOB SUB_TYPE 1,  
     LANGUAGE_REQ    VARCHAR(15) [1:5],  
   ```
4. Creating the PROJECT table with primary, foreign and unique key constraints with custom index names specified with the USING clause.

```sql
CREATE TABLE PROJECT (
    PROJ_ID     PROJNO NOT NULL,
    PROJ_NAME   VARCHAR(20) NOT NULL UNIQUE USING DESC INDEX IDX_PROJNAME,
    PROJ_DESC   BLOB SUB_TYPE 1,
    TEAM_LEADER EMPNO,
    PRODUCT     PRODTYPE,
    PRIMARY KEY (PROJ_ID) USING INDEX IDX_PROJ_ID,
    FOREIGN KEY (TEAM_LEADER) REFERENCES EMPLOYEE (EMP_NO)
);```

5. Creating a table with an identity column

```sql
create table objects (  
    id integer generated by default as identity primary key,
    name varchar(15)
);  
insert into objects (name) values ('Table');  
insert into objects (id, name) values (10, 'Computer');  
insert into objects (name) values ('Book');  
select * from objects order by id;  
```

```
<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Table</td>
</tr>
<tr>
<td>2</td>
<td>Book</td>
</tr>
<tr>
<td>10</td>
<td>Computer</td>
</tr>
</tbody>
</table>
```

6. Creating the SALARY_HISTORY table with two computed fields. The first one is declared according to the SQL:2003 standard, while the second one is declared according to the traditional declaration of computed fields in Firebird.

```sql
CREATE TABLE SALARY_HISTORY (  
    EMP_NO      EMPNO NOT NULL,
    CHANGE_DATE TIMESTAMP DEFAULT 'NOW' NOT NULL,
```
Global Temporary Tables (GTT)

Global temporary tables have persistent metadata, but their contents are transaction-bound (the default) or connection-bound. Every transaction or connection has its own private instance of a GTT, isolated from all the others. Instances are only created if and when the GTT is referenced. They are destroyed when the transaction ends or on disconnection. The metadata of a GTT can be modified or removed using `ALTER TABLE` and `DROP TABLE`, respectively.

**Syntax**

```sql
CREATE GLOBAL TEMPORARY TABLE tablename
    (<column_def> [, {<column_def> | <table_constraint>} ...])
ON COMMIT {DELETE | PRESERVE} ROWS
```

**Syntax notes**

- ON COMMIT DELETE ROWS creates a transaction-level GTT (the default), ON COMMIT PRESERVE ROWS a connection-level GTT
- An EXTERNAL [FILE] clause is not allowed in the definition of a global temporary table

Since Firebird 3.0, GTTs are writable in read-only transactions. The effect is as follows:

Read-only transaction in read-write database

  Writable in both ON COMMIT PRESERVE ROWS and ON COMMIT DELETE ROWS

Read-only transaction in read-only database

  Writable in ON COMMIT DELETE ROWS only

**Restrictions on GTTs**

GTTs can be “dressed up” with all the features and paraphernalia of ordinary tables (keys, references, indexes, triggers and so on) but there are a few restrictions:

- GTTs and regular tables cannot reference one another
- A connection-bound (“PRESERVE ROWS”) GTT cannot reference a transaction-bound (“DELETE ROWS”) GTT
- Domain constraints cannot reference any GTT
- The destruction of a GTT instance at the end of its life cycle does not cause any BEFORE/AFTER
delete triggers to fire

In an existing database, it is not always easy to distinguish a regular table from a GTT, or a transaction-level GTT from a connection-level GTT. Use this query to find out what type of table you are looking at:

```
select t.rdb$type_name
from rdb$relations r
join rdb$types t on r.rdb$relation_type = t.rdb$type
where t.rdb$field_name = 'RDB$RELATION_TYPE'
and r.rdb$relation_name = 'TABLENAME'
```

For an overview of the types of all the relations in the database:

```
select r.rdb$relation_name, t.rdb$type_name
from rdb$relations r
join rdb$types t on r.rdb$relation_type = t.rdb$type
where t.rdb$field_name = 'RDB$RELATION_TYPE'
and coalesce (r.rdb$system_flag, 0) = 0
```

The RDB$TYPE_NAME field will show PERSISTENT for a regular table, VIEW for a view, GLOBAL_TEMPORARY_PRESERVE for a connection-bound GTT and GLOBAL_TEMPORARY_DELETE for a transaction_bound GTT.

**Examples of Global Temporary Tables**


```
CREATE GLOBAL TEMPORARY TABLE MYCONNGTT (
    ID INTEGER NOT NULL PRIMARY KEY,
    TXT VARCHAR(32),
    TS TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) ON COMMIT PRESERVE ROWS;
```

2. Creating a transaction-scoped global temporary table that uses a foreign key to reference a connection-scoped global temporary table. The ON COMMIT sub-clause is optional because DELETE ROWS is the default.

```
CREATE GLOBAL TEMPORARY TABLE MYTXGTT (
    ID INTEGER NOT NULL PRIMARY KEY,
    PARENT_ID INTEGER NOT NULL REFERENCES MYCONNGTT(ID),
    TXT VARCHAR(32),
    TS TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) ON COMMIT DELETE ROWS;
```
External Tables

The optional EXTERNAL [FILE] clause specifies that the table is stored outside the database in an external text file of fixed-length records. The columns of a table stored in an external file can be of any type except BLOB or ARRAY, although for most purposes, only columns of CHAR types would be useful.

All you can do with a table stored in an external file is insert new rows (INSERT) and query the data (SELECT). Updating existing data (UPDATE) and deleting rows (DELETE) are not possible.

A file that is defined as an external table must be located on a storage device that is physically present on the machine where the Firebird server runs and, if the parameter ExternalFileAccess in the firebird.conf configuration file is Restrict, it must be in one of the directories listed there as the argument for Restrict. If the file does not exist yet, Firebird will create it on first access.

The ability to use external files for a table depends on the value set for the ExternalFileAccess parameter in firebird.conf:

- If it is set to None (the default), any attempt to access an external file will be denied.
- The Restrict setting is recommended, for restricting external file access to directories created explicitly for the purpose by the server administrator. For example:
  - ExternalFileAccess = Restrict externalfiles will restrict access to a directory named externalfiles directly beneath the Firebird root directory
  - ExternalFileAccess = d:\databases\outfiles; e:\infiles will restrict access to just those two directories on the Windows host server. Note that any path that is a network mapping will not work. Paths enclosed in single or double quotes will not work, either.
- If this parameter is set to Full, external files may be accessed anywhere on the host file system. This creates a security vulnerability and is not recommended.

External File Format

The “row” format of the external table is fixed length and binary. There are no field delimiters: both field and row boundaries are determined by maximum sizes, in bytes, of the field definitions. It is important to keep this in mind, both when defining the structure of the external table and when designing an input file for an external table that is to import data from another application. The ubiquitous “.csv” format, for example, is of no use as an input file and cannot be generated directly into an external file.

The most useful data type for the columns of external tables is the fixed-length CHAR type, of suitable lengths for the data they are to carry. Date and number types are easily cast to and from strings whereas, unless the files are to be read by another Firebird database, the native data types — binary data — will appear to external applications as unparsable “alphabetti”.

Of course, there are ways to manipulate typed data so as to generate output files from Firebird that can be read directly as input files to other applications, using stored procedures, with or without
employing external tables. Such techniques are beyond the scope of a language reference. Here, we provide some guidelines and tips for producing and working with simple text files, since the external table feature is often used as an easy way to produce or read transaction-independent logs that can be studied off-line in a text editor or auditing application.

**Row Delimiters**

Generally, external files are more useful if rows are separated by a delimiter, in the form of a “newline” sequence that is recognised by reader applications on the intended platform. For most contexts on Windows, it is the two-byte 'CRLF' sequence, carriage return (ASCII code decimal 13) and line feed (ASCII code decimal 10). On POSIX, LF on its own is usual; for some MacOSX applications, it may be LFCR. There are various ways to populate this delimiter column. In our example below, it is done by using a BEFORE INSERT trigger and the internal function ASCII_CHAR.

**External Table Example**

For our example, we will define an external log table that might be used by an exception handler in a stored procedure or trigger. The external table is chosen because the messages from any handled exceptions will be retained in the log, even if the transaction that launched the process is eventually rolled back because of another, unhandled exception. For demonstration purposes, it has just two data columns, a time stamp and a message. The third column stores the row delimiter:

```sql
CREATE TABLE ext_log
EXTERNAL FILE 'd:\externals\log_me.txt' (
    stamp CHAR (24),
    message CHAR(100),
    crlf CHAR(2) -- for a Windows context
);
COMMIT;
```

Now, a trigger, to write the timestamp and the row delimiter each time a message is written to the file:

```sql
SET TERM ^;
CREATE TRIGGER bi_ext_log FOR ext_log
ACTIVE BEFORE INSERT
AS
BEGIN
    IF (new.stamp is NULL) then
        new.stamp = CAST (CURRENT_TIMESTAMP as CHAR(24));
    new.crlf = ASCII_CHAR(13) || ASCII_CHAR(10);
END ^
COMMIT ^
SET TERM ;^;
```

Inserting some records (which could have been done by an exception handler or a fan of Shakespeare):
insert into ext_log (message)
values('Shall I compare thee to a summer''s day?');
insert into ext_log (message)
values('Thou art more lovely and more temperate');

The output:

2015-10-07 15:19:03.4110Shall I compare thee to a summer's day?
2015-10-07 15:19:58.7600Thou art more lovely and more temperate

5.4.2. ALTER TABLE

Used for
Altering the structure of a table.

Available in
DSQL, ESQL

Syntax

```
ALTER TABLE tablename
   <operation> [, <operation> ...]

<operation> ::= 
   ADD <col_def>
   | ADD tconstraint>
   | DROP colname
   | DROP CONSTRAINT constr_name
   | ALTER [COLUMN] colname <col_mod>

<col_def> ::= 
   <regular_col_def>
   | <computed_col_def>
   | <identity_col_def>

<regular_col_def> ::= 
   colname {<datatype> | domainname}
   [DEFAULT {<literal> | NULL | <context_var>}}
   [<col_constraint> ...]
   [COLLATE collation_name]

<computed_col_def> ::= 
   colname [{<datatype> | domainname}]
   {COMPUTED [BY] | GENERATED ALWAYS AS} (<expression>)

<identity_col_def> ::= 
   colname {<datatype> | domainname}
   GENERATED BY DEFAULT AS IDENTITY [(START WITH startvalue)]
```
<col_mod> ::= 
  TO newname  
  | POSITION newpos   
  | <regular_col_mod>  
  | <computed_col_mod>  
  | <identity_col_mod>

<regular_col_mod> ::= 
  TYPE {<datatype> | domainname} 
  | SET DEFAULT {<literal> | NULL | <context_var>} 
  | DROP DEFAULT 
  | {SET | DROP} NOT NULL

<computed_col_mod> ::= 
  [TYPE <datatype>] {COMPUTED [BY] | GENERATED ALWAYS AS} (<expression>)

<identity_col_mod> ::= 
  RESTART [WITH startvalue]

!! See CREATE TABLE syntax for further rules!!

Table 27. ALTER TABLE Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tablename</td>
<td>Name (identifier) of the table</td>
</tr>
<tr>
<td>operation</td>
<td>One of the available operations altering the structure of the table</td>
</tr>
<tr>
<td>colname</td>
<td>Name (identifier) for a column in the table, max. 31 characters. Must be</td>
</tr>
<tr>
<td></td>
<td>unique in the table.</td>
</tr>
<tr>
<td>domain_name</td>
<td>Domain name</td>
</tr>
<tr>
<td>newname</td>
<td>New name (identifier) for the column, max. 31 characters. Must be</td>
</tr>
<tr>
<td></td>
<td>unique in the table.</td>
</tr>
<tr>
<td>newpos</td>
<td>The new column position (an integer between 1 and the number of columns in</td>
</tr>
<tr>
<td></td>
<td>the table)</td>
</tr>
<tr>
<td>start_value</td>
<td>The first value of the identity column after restart</td>
</tr>
<tr>
<td>other_table</td>
<td>The name of the table referenced by the foreign key constraint</td>
</tr>
<tr>
<td>literal</td>
<td>A literal value that is allowed in the given context</td>
</tr>
<tr>
<td>context_var</td>
<td>A context variable whose type is allowed in the given context</td>
</tr>
<tr>
<td>check_condition</td>
<td>The condition of a CHECK constraint that will be satisfied if it evaluates to</td>
</tr>
<tr>
<td></td>
<td>TRUE or UNKNOWN/NULL</td>
</tr>
<tr>
<td>collation</td>
<td>Name of a collation sequence that is valid for charset_name, if it is</td>
</tr>
<tr>
<td></td>
<td>supplied with datatype or, otherwise, is valid for the default character</td>
</tr>
<tr>
<td></td>
<td>set of the database</td>
</tr>
</tbody>
</table>
The `ALTER TABLE` statement changes the structure of an existing table. With one `ALTER TABLE` statement it is possible to perform multiple operations, adding/dropping columns and constraints and also altering column specifications.

Multiple operations in an `ALTER TABLE` statement are separated with commas.

**Version Count Increments**

Some changes in the structure of a table increment the metadata change counter ("version count") assigned to every table. The number of metadata changes is limited to 255 for each table. Once the counter reaches the 255 limit, you will not be able to make any further changes to the structure of the table without resetting the counter.

**To reset the metadata change counter**

You need to back up and restore the database using the `gbak` utility.

**The ADD Clause**

With the `ADD` clause you can add a new column or a new table constraint. The syntax for defining the column and the syntax of defining the table constraint correspond with those described for `CREATE TABLE` statement.

**Effect on Version Count**

- Each time a new column is added, the metadata change counter is increased by one
- Adding a new table constraint does not increase the metadata change counter

**Points to Be Aware of**

1. Adding a column with a `NOT NULL` constraint without a `DEFAULT` value will—since Firebird 3.0—fail if the table has existing rows. When adding a non-nullable column, it is recommended either to set a default value for it, or to create it as nullable, update the column in existing rows with a non-null value, and then add a `NOT NULL` constraint.

2. When a new `CHECK` constraint is added, existing data is not tested for compliance. Prior testing of existing data against the new `CHECK` expression is recommended.

3. Although adding an identity column is supported, this will only succeed if the table is empty. Adding an identity column will fail if the table has one or more rows.

**The DROP Clause**

The `DROP colname` clause deletes the specified column from the table. An attempt to drop a column will fail if anything references it. Consider the following items as sources of potential dependencies:

- column or table constraints
• indexes
• stored procedures and triggers
• views

Effect on Version Count
• Each time a column is dropped, the table’s metadata change counter is increased by one.

The DROP CONSTRAINT Clause

The DROP CONSTRAINT clause deletes the specified column-level or table-level constraint.

A PRIMARY KEY or UNIQUE key constraint cannot be deleted if it is referenced by a FOREIGN KEY constraint in another table. It will be necessary to drop that FOREIGN KEY constraint before attempting to drop the PRIMARY KEY or UNIQUE key constraint it references.

Effect on Version Count
• Deleting a column constraint or a table constraint does not increase the metadata change counter.

The ALTER [COLUMN] Clause

With the ALTER [COLUMN] clause, attributes of existing columns can be modified without the need to drop and re-add the column. Permitted modifications are:

• change the name (does not affect the metadata change counter)
• change the data type (increases the metadata change counter by one)
• change the column position in the column list of the table (does not affect the metadata change counter)
• delete the default column value (does not affect the metadata change counter)
• set a default column value or change the existing default (does not affect the metadata change counter)
• change the type and expression for a computed column (does not affect the metadata change counter)
• set the NOT NULL constraint (does not affect the metadata change counter)
• drop the NOT NULL constraint (does not affect the metadata change counter)

Renaming a Column: the TO Clause

The TO keyword with a new identifier renames an existing column. The table must not have an existing column that has the same identifier.

It will not be possible to change the name of a column that is included in any constraint: PRIMARY KEY, UNIQUE key, FOREIGN KEY, column constraint or the CHECK constraint of the table.

Renaming a column will also be disallowed if the column is used in any trigger, stored procedure or view.
Changing the Data Type of a Column: the **TYPE** Clause

The keyword **TYPE** changes the data type of an existing column to another, allowable type. A type change that might result in data loss will be disallowed. As an example, the number of characters in the new type for a **CHAR** or **VARCHAR** column cannot be smaller than the existing specification for it.

If the column was declared as an array, no change to its type or its number of dimensions is permitted.

The data type of a column that is involved in a foreign key, primary key or unique constraint cannot be changed at all.

Changing the Position of a Column: the **POSITION** Clause

The **POSITION** keyword changes the position of an existing column in the notional “left-to-right” layout of the record.

Numbering of column positions starts at 1.

- If a position less than 1 is specified, an error message will be returned
- If a position number is greater than the number of columns in the table, its new position will be adjusted silently to match the number of columns.

The DROP DEFAULT and SET DEFAULT Clauses

The optional **DROP DEFAULT** clause deletes the default value for the column if it was put there previously by a **CREATE TABLE** or **ALTER TABLE** statement.

- If the column is based on a domain with a default value, the default value will revert to the domain default
- An execution error will be raised if an attempt is made to delete the default value of a column which has no default value or whose default value is domain-based

The optional **SET DEFAULT** clause sets a default value for the column. If the column already has a default value, it will be replaced with the new one. The default value applied to a column always overrides one inherited from a domain.

The SET NOT NULL and DROP NOT NULL Clauses

The **SET NOT NULL** clause adds a **NOT NULL** constraint on an existing table column. Contrary to definition in **CREATE TABLE**, it is not possible to specify a constraint name.

The successful addition of the **NOT NULL** constraint is subject to a full data validation on the table, so ensure that the column has no nulls before attempting the change.

An explicit **NOT NULL** constraint on domain-based column overrides domain settings. In this scenario, changing the domain to be nullable does not extend to a table column.

Dropping the **NOT NULL** constraint from the column if its type is a domain that also has a **NOT NULL**
The **computed** [by] or **generated** *always as* Clauses

The data type and expression underlying a computed column can be modified using a **computed** [by] or **generated always as** clause in the **alter table alter [column]** statement. Converting a regular column to a computed one and vice versa are not permitted.

**Changing Identity Columns**

For identity columns (**generated by default as identity**), it is possible to restart the sequence used for generating identity values. If only the **restart** clause is specified, then the sequence resets to the initial value specified during **create table**. If the optional **with start_value** clause is specified, the sequence will restart with the specified value.

It is not possible to convert an existing column to an identity column, or to convert an identity column to a normal column. Firebird 4 will introduce the ability to convert an identity column to a normal column.

Restarting is currently subject to a bug: the first value generated after a restart is 1 (one) higher than the configured initial value (or the value specified through **with**). See also **Identity Columns (autoincrement)**.

**Attributes that Cannot Be Altered**

The following alterations are not supported:

- Changing the collation of a character type column

**Who Can Alter a Table?**

The **alter table** statement can be executed by:

- **Administrators**
- The owner of the table
- Users with the **alter any table** privilege

**Examples Using **alter table**

1. Adding the **capital** column to the **country** table.

   ```sql
   alter table country
   add capital varchar(25);
   ```

2. Adding the **capital** column with the **not null** and **unique** constraint and deleting the **currency** column.
ALTER TABLE COUNTRY
   ADD CAPITAL VARCHAR(25) NOT NULL UNIQUE,
   DROP CURRENCY;

3. Adding the CHK_SALARY check constraint and a foreign key to the JOB table.

ALTER TABLE JOB
   ADD CONSTRAINT CHK_SALARY CHECK (MIN_SALARY < MAX_SALARY),
   ADD FOREIGN KEY (JOB_COUNTRY) REFERENCES COUNTRY (COUNTRY);

4. Setting default value for the MODEL field, changing the type of the ITEMID column and renaming the MODELNAME column.

ALTER TABLE STOCK
   ALTER COLUMN MODEL SET DEFAULT 1,
   ALTER COLUMN ITEMID TYPE BIGINT,
   ALTER COLUMN MODELNAME TO NAME;

5. Restarting the sequence of an identity column.

ALTER TABLE objects
   ALTER ID RESTART WITH 100;

6. Changing the computed columns NEW_SALARY and SALARY_CHANGE.

ALTER TABLE SALARY_HISTORY
   ALTER NEW_SALARY GENERATED ALWAYS AS
   (OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100),
   ALTER SALARY_CHANGE COMPUTED BY
   (OLD_SALARY * PERCENT_CHANGE / 100);

See also
CREATE TABLE, DROP TABLE, CREATE DOMAIN

5.4.3. DROP TABLE

Used for
Dropping (deleting) a table

Available in
DSQL, ESQL
Syntax

DROP TABLE tablename

Table 28. DROP TABLE Statement Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tablename</td>
<td>Name (identifier) of the table</td>
</tr>
</tbody>
</table>

The `DROP TABLE` statement drops (deletes) an existing table. If the table has dependencies, the `DROP TABLE` statement will fail with an execution error.

When a table is dropped, all its triggers and indexes will be deleted as well.

**Who Can Drop a Table?**

The `DROP TABLE` statement can be executed by:

- **Administrators**
- The owner of the table
- Users with the `DROP ANY TABLE` privilege

**Example of DROP TABLE**

*Dropping the COUNTRY table.*

```
DROP TABLE COUNTRY;
```

See also

`CREATE TABLE, ALTER TABLE, RECREATE TABLE`

### 5.4.4. RECREATE TABLE

*Used for*

Creating a new table (relation) or recreating an existing one

*Available in*

DSQL

**Syntax**

RECREATE [GLOBAL TEMPORARY] TABLE tablename
[EXTERNAL [FILE] 'filespec']
(<col_def> [, {<col_def> | <tconstraint>} ...])
[ON COMMIT {DELETE | PRESERVE} ROWS]

See the `CREATE TABLE section` for the full syntax of `CREATE TABLE` and descriptions of defining tables, columns and constraints.
RECREATE TABLE creates or recreates a table. If a table with this name already exists, the RECREATE TABLE statement will try to drop it and create a new one. Existing dependencies will prevent the statement from executing.

**Example of RECREATE TABLE**

Creating or recreating the COUNTRY table.

```sql
RECREATE TABLE COUNTRY (  
    COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,  
    CURRENCY VARCHAR(10) NOT NULL  
);
```

See also
CREATE TABLE, DROP TABLE

## 5.5. INDEX

An index is a database object used for faster data retrieval from a table or for speeding up the sorting in a query. Indexes are used also to enforce the referential integrity constraints PRIMARY KEY, FOREIGN KEY and UNIQUE.

This section describes how to create indexes, activate and deactivate them, delete them and collect statistics (recalculate selectivity) for them.

### 5.5.1. CREATE INDEX

*Used for*
Creating an index for a table

*Available in*
DSQL, ESQL

*Syntax*

```sql
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]] INDEX indexname ON tablename 
{((col [, col ...]) | COMPUTED BY (<expression>))
```

*Table 29. CREATE INDEX Statement Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexname</td>
<td>Index name. It may consist of up to 31 characters</td>
</tr>
<tr>
<td>tablename</td>
<td>The name of the table for which the index is to be built</td>
</tr>
<tr>
<td>col</td>
<td>Name of a column in the table. Columns of the types BLOB and ARRAY and computed fields cannot be used in an index</td>
</tr>
</tbody>
</table>
The `CREATE INDEX` statement creates an index for a table that can be used to speed up searching, sorting and grouping. Indexes are created automatically in the process of defining constraints, such as primary key, foreign key or unique constraints.

An index can be built on the content of columns of any data type except for `BLOB` and arrays. The name (identifier) of an index must be unique among all index names.

### Key Indexes

When a primary key, foreign key or unique constraint is added to a table or column, an index with the same name is created automatically, without an explicit directive from the designer. For example, the `PK_COUNTRY` index will be created automatically when you execute and commit the following statement:

```
ALTER TABLE COUNTRY ADD CONSTRAINT PK_COUNTRY
  PRIMARY KEY (ID);
```

### Who Can Create an Index?

The `CREATE INDEX` statement can be executed by:

- Administrators
- The owner of the table
- Users with the `ALTER ANY TABLE` privilege

### Unique Indexes

Specifying the keyword `UNIQUE` in the index creation statement creates an index in which uniqueness will be enforced throughout the table. The index is referred to as a “unique index”. A unique index is not a constraint.

Unique indexes cannot contain duplicate key values (or duplicate key value combinations, in the case of compound, or multi-column, or multi-segment) indexes. Duplicated NULLs are permitted, in accordance with the SQL:99 standard, in both single-segment and multi-segment indexes.

### Index Direction

All indexes in Firebird are uni-directional. An index may be constructed from the lowest value to the highest (ascending order) or from the highest value to the lowest (descending order). The keywords `ASC[ENDING]` and `DESC[ENDING]` are used to specify the direction of the index. The default index order is `ASC[ENDING]`. It is quite valid to define both an ascending and a descending index on the same column or key set.

A descending index can be useful on a column that will be subjected to searches on
the high values (“newest”, maximum, etc.)

Firebird uses B-tree indexes, which are bidirectional. However, due to technical limitations, Firebird uses an index in one direction only.

See also Firebird for the Database Expert: Episode 3 - On disk consistency

**Computed (Expression) Indexes**

In creating an index, you can use the `COMPUTED BY` clause to specify an expression instead of one or more columns. Computed indexes are used in queries where the condition in a `WHERE`, `ORDER BY` or `GROUP BY` clause exactly matches the expression in the index definition. The expression in a computed index may involve several columns in the table.

Expression indexes can also be used as a workaround for indexing computed columns: use the name of the computed column as the expression.

**Limits on Indexes**

Certain limits apply to indexes.

The maximum length of a key in an index is limited to $\frac{1}{4}$ of the page size.

**Maximum Indexes per Table**

The number of indexes that can be accommodated for each table is limited. The actual maximum for a specific table depends on the page size and the number of columns in the indexes.

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Number of Indexes Depending on Column Count</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single</strong></td>
<td><strong>2-Column</strong></td>
</tr>
<tr>
<td>4096</td>
<td>203</td>
</tr>
<tr>
<td>8192</td>
<td>408</td>
</tr>
<tr>
<td>16384</td>
<td>818</td>
</tr>
</tbody>
</table>

**Character Index Limits**

The maximum indexed string length is 9 bytes less than the maximum key length. The maximum indexable string length depends on the page size and the character set.

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Maximum Indexable String Length by Charset Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 byte/char</strong></td>
<td><strong>2 byte/char</strong></td>
</tr>
<tr>
<td>4096</td>
<td>1015</td>
</tr>
<tr>
<td>8192</td>
<td>2039</td>
</tr>
</tbody>
</table>
Examples Using `CREATE INDEX`

1. Creating an index for the `UPDATER_ID` column in the `SALARY_HISTORY` table

   ```sql
   CREATE INDEX IDX_UPDATER ON SALARY_HISTORY (UPDATER_ID);
   ```

2. Creating an index with keys sorted in the descending order for the `CHANGE_DATE` column in the `SALARY_HISTORY` table

   ```sql
   CREATE DESCENDING INDEX IDX_CHANGE ON SALARY_HISTORY (CHANGE_DATE);
   ```

3. Creating a multi-segment index for the `ORDER_STATUS`, `PAID` columns in the `SALES` table

   ```sql
   CREATE INDEX IDX_SALESTAT ON SALES (ORDER_STATUS, PAID);
   ```

4. Creating an index that does not permit duplicate values for the `NAME` column in the `COUNTRY` table

   ```sql
   CREATE UNIQUE INDEX UNQ_COUNTRY_NAME ON COUNTRY (NAME);
   ```

5. Creating a computed index for the `PERSONS` table

   ```sql
   CREATE INDEX IDX_NAME_UPPER ON PERSONS COMPUTED BY (UPPER (NAME));
   ```

   An index like this can be used for a case-insensitive search:

   ```sql
   SELECT * FROM PERSONS WHERE UPPER(NAME) STARTING WITH UPPER('Iv');
   ```

See also

`ALTER INDEX`, `DROP INDEX`

### 5.5.2. ALTER INDEX

*Used for*

Activating or deactivating an index; rebuilding an index
Available in
DSQL, ESQL

Syntax

```
ALTER INDEX indexname {ACTIVE | INACTIVE}
```

Table 32. ALTER INDEX Statement Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexname</td>
<td>Index name</td>
</tr>
</tbody>
</table>

The ALTER INDEX statement activates or deactivates an index. There is no facility on this statement for altering any attributes of the index.

**INACTIVE**

With the INACTIVE option, the index is switched from the active to inactive state. The effect is similar to the DROP INDEX statement except that the index definition remains in the database. Altering a constraint index to the inactive state is not permitted.

An active index can be deactivated if there are no queries prepared using that index; otherwise, an “object in use” error is returned.

Activating an inactive index is also safe. However, if there are active transactions modifying the table, the transaction containing the ALTER INDEX statement will fail if it has the NOWAIT attribute. If the transaction is in WAIT mode, it will wait for completion of concurrent transactions.

On the other side of the coin, if our ALTER INDEX succeeds and starts to rebuild the index at COMMIT, other transactions modifying that table will fail or wait, according to their WAIT/NO WAIT attributes. The situation is exactly the same for CREATE INDEX.

**How is it Useful?**

It might be useful to switch an index to the inactive state whilst inserting, updating or deleting a large batch of records in the table that owns the index.

**ACTIVE**

With the ACTIVE option, if the index is in the inactive state, it will be switched to active state and the system rebuilds the index.

**How is it Useful?**

Even if the index is active when ALTER INDEX … ACTIVE is executed, the index will be rebuilt. Rebuilding indexes can be a useful piece of housetaking to do, occasionally, on the indexes of a large table in a database that has frequent inserts, updates or deletes but is infrequently restored.

**Who Can Alter an Index?**

The ALTER INDEX statement can be executed by:
• Administrators
• The owner of the table
• Users with the ALTER ANY TABLE privilege

Use of ALTER INDEX on a Constraint Index

Altering the index of a PRIMARY KEY, FOREIGN KEY or UNIQUE constraint to INACTIVE is not permitted. However, ALTER INDEX … ACTIVE works just as well with constraint indexes as it does with others, as an index rebuilding tool.

ALTER INDEX Examples

1. Deactivating the IDX_UPDATER index

   ```sql
   ALTER INDEX IDX_UPDATER INACTIVE;
   ```

2. Switching the IDX_UPDATER index back to the active state and rebuilding it

   ```sql
   ALTER INDEX IDX_UPDATER ACTIVE;
   ```

See also
CREATE INDEX, DROP INDEX, SET STATISTICS

5.5.3. DROP INDEX

Used for
Dropping (deleting) an index

Available in
DSQL, ESQL

Syntax

```sql
DROP INDEX indexname
```  

Table 33. DROP INDEX Statement Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexname</td>
<td>Index name</td>
</tr>
</tbody>
</table>

The DROP INDEX statement drops (deletes) the named index from the database.

A constraint index cannot dropped using DROP INDEX. Constraint indexes are dropped during the process of executing the command ALTER TABLE … DROP CONSTRAINT ….
**Who Can Drop an Index?**

The `DROP INDEX` statement can be executed by:

- **Administrators**
- The owner of the table
- Users with the `ALTER ANY TABLE` privilege

**DROP INDEX Example**

_Dropping the IDX_UPDATER index_

```sql
DROP INDEX IDX_UPDATER;
```

*See also*

`CREATE INDEX`, `ALTER INDEX`

### 5.5.4. SET STATISTICS

**Used for**
Recalculating the selectivity of an index

**Available in**
DSQL, ESQL

**Syntax**

```
SET STATISTICS INDEX indexname
```

**Table 34. SET STATISTICS Statement Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexname</td>
<td>Index name</td>
</tr>
</tbody>
</table>

The `SET STATISTICS` statement recalculates the selectivity of the specified index.

**Who Can Update Index Statistics?**

The `SET STATISTICS` statement can be executed by:

- **Administrators**
- The owner of the table
- Users with the `ALTER ANY TABLE` privilege

**Index Selectivity**

The selectivity of an index is the result of evaluating the number of rows that can be selected in a search on every index value. A unique index has the maximum selectivity because it is impossible
to select more than one row for each value of an index key if it is used. Keeping the selectivity of an index up to date is important for the optimizer’s choices in seeking the most optimal query plan.

Index statistics in Firebird are not automatically recalculated in response to large batches of inserts, updates or deletions. It may be beneficial to recalculate the selectivity of an index after such operations because the selectivity tends to become outdated.

The statements CREATE INDEX and ALTER INDEX ACTIVE both store index statistics that completely correspond to the contents of the newly-[re]built index.

It can be performed under concurrent load without risk of corruption. However, be aware that, under concurrent load, the newly calculated statistics could become outdated as soon as SET STATISTICS finishes.

Example Using SET STATISTICS

Recalculating the selectivity of the index IDX_UPDATER

```
SET STATISTICS INDEX IDX_UPDATER;
```

See also

CREATE INDEX, ALTER INDEX

5.6. VIEW

A view is a virtual table that is actually a stored and named SELECT query for retrieving data of any complexity. Data can be retrieved from one or more tables, from other views and also from selectable stored procedures.

Unlike regular tables in relational databases, a view is not an independent data set stored in the database. The result is dynamically created as a data set when the view is selected.

The metadata of a view are available to the process that generates the binary code for stored procedures and triggers, just as though they were concrete tables storing persistent data.

5.6.1. CREATE VIEW

Used for

Creating a view

Available in

DSQL

Syntax

```
CREATE VIEW viewname [ <full_column_list> ]
AS <select_statement>
[ WITH CHECK OPTION ]
```
The CREATE VIEW statement creates a new view. The identifier (name) of a view must be unique among the names of all views, tables and stored procedures in the database.

The name of the new view can be followed by the list of column names that should be returned to the caller when the view is invoked. Names in the list do not have to be related to the names of the columns in the base tables from which they derive.

If the view column list is omitted, the system will use the column names and/or aliases from the SELECT statement. If duplicate names or non-aliased expression-derived columns make it impossible to obtain a valid list, creation of the view fails with an error.

The number of columns in the view's list must exactly match the number of columns in the selection list of the underlying SELECT statement in the view definition.

### Additional Points

- If the full list of columns is specified, it makes no sense to specify aliases in the SELECT statement because the names in the column list will override them
- The column list is optional if all the columns in the SELECT are explicitly named and are unique in the selection list

### Updatable Views

A view can be updatable or read-only. If a view is updatable, the data retrieved when this view is called can be changed by the DML statements INSERT, UPDATE, DELETE, UPDATE OR INSERT or MERGE. Changes made in an updatable view are applied to the underlying table(s).

A read-only view can be made updateable with the use of triggers. Once triggers have been defined on a view, changes posted to it will never be written automatically to the underlying table, even if the view was updatable to begin with. It is the responsibility of the programmer to ensure that the triggers update (or delete from, or insert into) the base tables as needed.

A view will be automatically updatable if all the following conditions are met:

- the SELECT statement queries only one table or one updatable view
- the SELECT statement does not call any stored procedures
- each base table (or base view) column not present in the view definition meets one of the...
following conditions:
◦ it is nullable
◦ it has a non-NULL default value
◦ it has a trigger that supplies a permitted value

• the SELECT statement contains no fields derived from subqueries or other expressions
• the SELECT statement does not contain fields defined through aggregate functions (MIN, MAX, AVG, SUM, COUNT, LIST, etc.), statistical functions (CORR, COVAR_POP, COVAR_SAMP, etc.), linear regression functions (REGR_AVGX, REGR_AVGY, etc.) or any type of window function
• the SELECT statement contains no ORDER BY, GROUP BY or HAVING clause
• the SELECT statement does not include the keyword DISTINCT or row-restrictive keywords such as ROWS, FIRST, SKIP, OFFSET or FETCH

### WITH CHECK OPTION

The optional `WITH CHECK OPTION` clause requires an updatable view to check whether new or updated data meet the condition specified in the `WHERE` clause of the `SELECT` statement. Every attempt to insert a new record or to update an existing one is checked whether the new or updated record would meet the `WHERE` criteria. If they fail the check, the operation is not performed and an appropriate error message is returned.

`WITH CHECK OPTION` can be specified only in a `CREATE VIEW` statement in which a `WHERE` clause is present to restrict the output of the main `SELECT` statement. An error message is returned otherwise.

**Please note:**

If `WITH CHECK OPTION` is used, the engine checks the input against the `WHERE` clause before passing anything to the base relation. Therefore, if the check on the input fails, any default clauses or triggers on the base relation that might have been designed to correct the input will never come into action.

Furthermore, view fields omitted from the `INSERT` statement are passed as NULLs to the base relation, regardless of their presence or absence in the `WHERE` clause. As a result, base table defaults defined on such fields will not be applied. Triggers, on the other hand, will fire and work as expected.

For views that do not have `WITH CHECK OPTION`, fields omitted from the `INSERT` statement are not passed to the base relation at all, so any defaults will be applied.

### Who Can Create a View?

The `CREATE VIEW` statement can be executed by:

- **Administrators**
- **Users with the `CREATE VIEW` privilege**

The creator of a view becomes its owner.
To create a view, a non-admin user also needs at least SELECT access to the underlying table(s) and/or view(s), and the EXECUTE privilege on any selectable stored procedures involved.

To enable insertions, updates and deletions through the view, the creator/owner must also possess the corresponding INSERT, UPDATE and DELETE rights on the underlying object(s).

Granting other users privileges on the view is only possible if the view owner has these privileges on the underlying objects WITH GRANT OPTION. It will always be the case if the view owner is also the owner of the underlying objects.

**Examples of Creating Views**

1. Creating view returning the JOB_CODE and JOB_TITLE columns only for those jobs where MAX_SALARY is less than $15,000.

   ```sql
   CREATE VIEW ENTRY_LEVEL_JOBS AS
   SELECT JOB_CODE, JOB_TITLE
   FROM JOB
   WHERE MAX_SALARY < 15000;
   ```

2. Creating a view returning the JOB_CODE and JOB_TITLE columns only for those jobs where MAX_SALARY is less than $15,000. Whenever a new record is inserted or an existing record is updated, the MAX_SALARY < 15000 condition will be checked. If the condition is not true, the insert/update operation will be rejected.

   ```sql
   CREATE VIEW ENTRY_LEVEL_JOBS AS
   SELECT JOB_CODE, JOB_TITLE
   FROM JOB
   WHERE MAX_SALARY < 15000
   WITH CHECK OPTION;
   ```

3. Creating a view with an explicit column list.

   ```sql
   CREATE VIEW PRICE_WITH_MARKUP (CODE_PRICE, COST, COST_WITH_MARKUP) AS
   SELECT CODE_PRICE, COST, COST * 1.1
   FROM PRICE;
   ```

4. Creating a view with the help of aliases for fields in the SELECT statement (the same result as in Example 3).
CREATE VIEW PRICE_WITH_MARKUP AS
  SELECT
    CODE_PRICE,
    COST,
    COST * 1.1 AS COST_WITH_MARKUP
  FROM PRICE;

5. Creating a read-only view based on two tables and a stored procedure.

CREATE VIEW GOODS_PRICE AS
  SELECT
    goods.name AS goodsname,
    price.cost AS cost,
    b.quantity AS quantity
  FROM
    goods
    JOIN price ON goods.code_goods = price.code_goods
    LEFT JOIN sp_get_balance(goods.code_goods) b ON 1 = 1;

See also

ALTER VIEW, CREATE OR ALTER VIEW, RECREATE VIEW, DROP VIEW

5.6.2. ALTER VIEW

Used for

Modifying an existing view

Available in

DSQL

Syntax

ALTER VIEW viewname [<full_column_list>]
  AS <select_statement>
  [WITH CHECK OPTION]

<full_column_list> ::= (colname [, colname ...])

Table 36. ALTER VIEW Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>viewname</td>
<td>Name of an existing view</td>
</tr>
<tr>
<td>select_statement</td>
<td>SELECT statement</td>
</tr>
<tr>
<td>full_column_list</td>
<td>The list of columns in the view</td>
</tr>
<tr>
<td>colname</td>
<td>View column name. Duplicate column names are not allowed.</td>
</tr>
</tbody>
</table>
Use the `ALTER VIEW` statement for changing the definition of an existing view. Privileges for views remain intact and dependencies are not affected.

The syntax of the `ALTER VIEW` statement corresponds completely with that of `CREATE VIEW`.

Be careful when you change the number of columns in a view. Existing application code and PSQL modules that access the view may become invalid. For information on how to detect this kind of problem in stored procedures and trigger, see *The RDB$VALID_BLR Field* in the Appendix.

**Who Can Alter a View?**

The `ALTER VIEW` statement can be executed by:

- Administrators
- The owner of the view
- Users with the `ALTER ANY VIEW` privilege

**Example using `ALTER VIEW`**

*Altering the view `PRICE_WITH_MARKUP`*

```sql
ALTER VIEW PRICE_WITH_MARKUP (
    CODE_PRICE,
    COST,
    COST_WITH_MARKUP
) AS
SELECT
    CODE_PRICE,
    COST,
    COST * 1.15
FROM PRICE;
```

*See also*

CREATE VIEW, CREATE OR ALTER VIEW, RECREATE VIEW

**5.6.3. CREATE OR ALTER VIEW**

*Used for*

Creating a new view or altering an existing view.

*Available in*

DSQL

*Syntax*

```
CREATE OR ALTER VIEW viewname [<full_column_list>]
AS <select_statement>
```
Table 37. CREATE OR ALTER VIEW Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>viewname</td>
<td>Name of a view which may or may not exist</td>
</tr>
<tr>
<td>select_statement</td>
<td>SELECT statement</td>
</tr>
<tr>
<td>full_column_list</td>
<td>The list of columns in the view</td>
</tr>
<tr>
<td>colname</td>
<td>View column name. Duplicate column names are not allowed.</td>
</tr>
</tbody>
</table>

Use the CREATE OR ALTER VIEW statement for changing the definition of an existing view or creating it if it does not exist. Privileges for an existing view remain intact and dependencies are not affected.

The syntax of the CREATE OR ALTER VIEW statement corresponds completely with that of CREATE VIEW.

**Example of CREATE OR ALTER VIEW**

*Creating the new view PRICE_WITH_MARKUP view or altering it if it already exists*

```sql
CREATE OR ALTER VIEW PRICE_WITH_MARKUP (  
    CODE_PRICE,  
    COST,  
    COST_WITH_MARKUP  
) AS  
SELECT  
    CODE_PRICE,  
    COST,  
    COST * 1.15  
FROM PRICE;
```

See also

CREATE VIEW, ALTER VIEW, RECREATE VIEW

5.6.4. DROP VIEW

*Used for*

Deleting (dropping) a view

*Available in*

DSQL

*Syntax*

```sql
DROP VIEW viewname
```
Table 38. DROP VIEW Statement Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>viewname</td>
<td>View name</td>
</tr>
</tbody>
</table>

The DROP VIEW statement drops (deletes) an existing view. The statement will fail if the view has dependencies.

**Who Can Drop a View?**

The DROP VIEW statement can be executed by:

- Administrators
- The owner of the view
- Users with the DROP ANY VIEW privilege

**Example**

Deleting the PRICE_WITH_MARKUP view

```
DROP VIEW PRICE_WITH_MARKUP;
```

See also

CREATE VIEW, RECREATE VIEW, CREATE OR ALTER VIEW

5.6.5. RECREATE VIEW

**Used for**
Creating a new view or recreating an existing view

**Available in**
DSQL

**Syntax**

```
RECREATE VIEW viewname [full_column_list]
   AS select_statement
   [WITH CHECK OPTION]
   <full_column_list> ::= (colname [, colname ...])
```

Table 39. RECREATE VIEW Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>viewname</td>
<td>View name, maximum 31 characters</td>
</tr>
<tr>
<td>select_statement</td>
<td>SELECT statement</td>
</tr>
<tr>
<td>full_column_list</td>
<td>The list of columns in the view</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>colname</td>
<td>View column name. Duplicate column names are not allowed.</td>
</tr>
</tbody>
</table>

Creates or recreates a view. If there is a view with this name already, the engine will try to drop it before creating the new instance. If the existing view cannot be dropped, because of dependencies or insufficient rights, for example, `RECREATE VIEW` fails with an error.

**Example of RECREATE VIEW**

Creating the new view `PRICE_WITH_MARKUP` view or recreating it, if it already exists

```sql
RECREATE VIEW PRICE_WITH_MARKUP (
  CODE_PRICE,
  COST,
  COST_WITH_MARKUP
) AS
SELECT
  CODE_PRICE,
  COST,
  COST * 1.15
FROM PRICE;
```

See also

CREATE VIEW, DROP VIEW, CREATE OR ALTER VIEW

## 5.7. TRIGGER

A trigger is a special type of stored procedure that is not called directly, instead being executed when a specified event occurs in the associated table or view. A DML trigger is specific to one and only one relation (table or view) and one phase in the timing of the event (BEFORE or AFTER). It can be specified to execute for one specific event (insert, update, delete) or for some combination of two or three of those events.

Two other forms of trigger exist:

1. a “database trigger” can be specified to fire at the start or end of a user session (connection) or a user transaction.
2. a “DDL trigger” can be specified to fire at the before or after execution of one or more types of DDL statements.

### 5.7.1. CREATE TRIGGER

*Used for*

Creating a new trigger

*Available in*

DSQL, ESQL
Syntax

CREATE TRIGGER triname
   { <relation_trigger_legacy>
   | <relation_trigger_sql2003>
   | <database_trigger>
   | <ddl_trigger> }
<module-body>

<module-body> ::= !! See Syntax of Module Body !!

<relation_trigger_legacy> ::= FOR {tablename | viewname}
   [ACTIVE | INACTIVE]
   {BEFORE | AFTER} <mutation_list>
   [POSITION number]

<relation_trigger_sql2003> ::= [ACTIVE | INACTIVE]
   {BEFORE | AFTER} <mutation_list>
   [POSITION number]
   ON {tablename | viewname}

<database_trigger> ::= [ACTIVE | INACTIVE] ON <db_event>
   [POSITION number]

<ddl_trigger> ::= [ACTIVE | INACTIVE]
   {BEFORE | AFTER} <ddl_event>
   [POSITION number]

<mutation_list> ::= <mutation> [OR <mutation> [OR <mutation>]]

<mutation> ::= INSERT | UPDATE | DELETE

<db_event> ::= CONNECT | DISCONNECT
   | TRANSACTION {START | COMMIT | ROLLBACK}

<ddl_event> ::= ANY DDL STATEMENT
   | <ddl_event_item> [{OR <ddl_event_item>} ...]

<ddl_event_item> ::= {CREATE | ALTER | DROP} TABLE
   | {CREATE | ALTER | DROP} PROCEDURE
   | {CREATE | ALTER | DROP} FUNCTION
   | {CREATE | ALTER | DROP} TRIGGER
Table 40. CREATE TRIGGER Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trigname</td>
<td>Trigger name consisting of up to 31 characters. It must be unique among all trigger names in the database.</td>
</tr>
<tr>
<td>relation_trigger_legacy</td>
<td>Legacy style of trigger declaration for a relation trigger</td>
</tr>
<tr>
<td>relation_trigger_sql2003</td>
<td>Relation trigger declaration compliant with the SQL:2003 standard</td>
</tr>
<tr>
<td>database_trigger</td>
<td>Database trigger declaration</td>
</tr>
<tr>
<td>tablename</td>
<td>Name of the table with which the relation trigger is associated</td>
</tr>
<tr>
<td>viewname</td>
<td>Name of the view with which the relation trigger is associated</td>
</tr>
<tr>
<td>mutation_list</td>
<td>List of relation (table</td>
</tr>
<tr>
<td>number</td>
<td>Position of the trigger in the firing order. From 0 to 32,767</td>
</tr>
<tr>
<td>db_event</td>
<td>Connection or transaction event</td>
</tr>
<tr>
<td>ddl_event</td>
<td>List of metadata change events</td>
</tr>
<tr>
<td>ddl_event_item</td>
<td>One of the metadata change events</td>
</tr>
</tbody>
</table>

The CREATE TRIGGER statement is used for creating a new trigger. A trigger can be created either for a relation (table | view) event (or a combination of events), for a database event, or for a DDL event.

CREATE TRIGGER, along with its associates ALTER TRIGGER, CREATE OR ALTER TRIGGER and RECREATE TRIGGER, is a compound statement, consisting of a header and a body. The header specifies the name of the trigger, the name of the relation (for a DML trigger), the phase of the trigger, the event(s) it applies to, and the position to determine an order between triggers.

The trigger body consists of optional declarations of local variables and named cursors followed by one or more statements, or blocks of statements, all enclosed in an outer block that begins with the keyword BEGIN and ends with the keyword END. Declarations and embedded statements are terminated with semi-colons (‘;’).

The name of the trigger must be unique among all trigger names.
Statement Terminators

Some SQL statement editors—specifically the isql utility that comes with Firebird and possibly some third-party editors—employ an internal convention that requires all statements to be terminated with a semi-colon. This creates a conflict with PSQL syntax when coding in these environments. If you are unacquainted with this problem and its solution, please study the details in the PSQL chapter in the section entitled Switching the Terminator in isql.

External UDR Triggers

A trigger can also be located in an external module. In this case, instead of a trigger body, the CREATE TRIGGER specifies the location of the trigger in the external module using the EXTERNAL clause. The optional NAME clause specifies the name of the external module, the name of the trigger inside the module, and—optionally—user-defined information. The required ENGINE clause specifies the name of the UDR engine that handles communication between Firebird and the external module. The optional AS clause accepts a string literal “body”, which can be used by the engine or module for various purposes.

DML Triggers (on Tables or Views)

DML—or “relation”—triggers are executed at the row (record) level, every time the row image changes. A trigger can be either ACTIVE or INACTIVE. Only active triggers are executed. Triggers are created ACTIVE by default.

Who Can Create a DML Trigger?

DML triggers can be created by:

• Administrators
• The owner of the table (or view)
• Users with the ALTER ANY TABLE or — for a view — ALTER ANY VIEW privilege

Forms of Declaration

Firebird supports two forms of declaration for relation triggers:

• The original, legacy syntax
• The SQL:2003 standard-compliant form (recommended)

The SQL:2003 standard-compliant form is the recommended one.

A relation trigger specifies — among other things — a phase and one or more events.

Phase

Phase concerns the timing of the trigger with regard to the change-of-state event in the row of data:

• A BEFORE trigger is fired before the specified database operation (insert, update or delete) is carried out
• An AFTER trigger is fired after the database operation has been completed
Row Events

A relation trigger definition specifies at least one of the DML operations INSERT, UPDATE and DELETE, to indicate one or more events on which the trigger should fire. If multiple operations are specified, they must be separated by the keyword OR. No operation may occur more than once.

Within the statement block, the Boolean context variables INSERTING, UPDATING and DELETING can be used to test which operation is currently executing.

Firing Order of Triggers

The keyword POSITION allows an optional execution order (“firing order”) to be specified for a series of triggers that have the same phase and event as their target. The default position is 0. If no positions are specified, or if several triggers have a single position number, the triggers will be executed in the alphabetical order of their names.

Variable Declarations

The optional declarations section beneath the keyword AS in the header of the trigger is for defining variables and named cursors that are local to the trigger. For more details, see DECLARE VARIABLE and DECLARE CURSOR in the Procedural SQL chapter.

The Trigger Body

The local declarations (if any) are the final part of a trigger’s header section. The trigger body follows, where one or more blocks of PSQL statements are enclosed in a structure that starts with the keyword BEGIN and terminates with the keyword END.

Only the owner of the view or table and administrators have the authority to use CREATE TRIGGER.

Examples of CREATE TRIGGER for Tables and Views

1. Creating a trigger in the “legacy” form, firing before the event of inserting a new record into the CUSTOMER table occurs.

   ```sql
   CREATE TRIGGER SET_CUST_NO FOR CUSTOMER
   ACTIVE BEFORE INSERT POSITION 0
   AS
   BEGIN
     IF (NEW.CUST_NO IS NULL) THEN
       NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
   END
   ```

2. Creating a trigger firing before the event of inserting a new record into the CUSTOMER table in the SQL:2003 standard-compliant form.

   ```sql
   CREATE TRIGGER set_cust_no
   ACTIVE BEFORE INSERT POSITION 0 ON customer
   AS
   BEGIN
   ```
IF (NEW.cust_no IS NULL) THEN
    NEW.cust_no = GEN_ID(cust_no_gen, 1);
END

3. Creating a trigger that will fire after either inserting, updating or deleting a record in the CUSTOMER table.

CREATE TRIGGER TR_CUST_LOG
ACTIVE AFTER INSERT OR UPDATE OR DELETE POSITION 10
ON CUSTOMER
AS
BEGIN
    INSERT INTO CHANGE_LOG (LOG_ID,
                            ID_TABLE,
                            TABLE_NAME,
                            MUTATION)
    VALUES (NEXT VALUE FOR SEQ_CHANGE_LOG,
            OLD.CUST_NO,
            'CUSTOMER',
            CASE
                WHEN INSERTING THEN 'INSERT'
                WHEN UPDATING THEN 'UPDATE'
                WHEN DELETING THEN 'DELETE'
            END);
END

Database Triggers

Triggers can be defined to fire upon “database events”, which really refers to a mixture of events that act across the scope of a session (connection) and events that act across the scope of an individual transaction:

- CONNECT
- DISCONNECT
- TRANSACTION START
- TRANSACTION COMMIT
- TRANSACTION ROLLBACK

DDL Triggers are a sub-type of database triggers, covered in a separate section.

Who Can Create a Database Trigger?

Database triggers can be created by:

- Administrators
- Users with the ALTER DATABASE privilege
Execution of Database Triggers and Exception Handling

CONNECT and DISCONNECT triggers are executed in a transaction created specifically for this purpose. This transaction uses the default isolation level, i.e. snapshot (concurrency), write and wait. If all goes well, the transaction is committed. Uncaught exceptions cause the transaction to roll back, and

- for a CONNECT trigger, the connection is then broken and the exception is returned to the client
- for a DISCONNECT trigger, exceptions are not reported. The connection is broken as intended

TRANSACTION triggers are executed within the transaction whose start, commit or rollback evokes them. The action taken after an uncaught exception depends on the event:

- In a TRANSACTION START trigger, the exception is reported to the client and the transaction is rolled back
- In a TRANSACTION COMMIT trigger, the exception is reported, the trigger’s actions so far are undone and the commit is cancelled
- In a TRANSACTION ROLLBACK trigger, the exception is not reported and the transaction is rolled back as intended.

Traps

Obviously there is no direct way of knowing if a DISCONNECT or TRANSACTION ROLLBACK trigger caused an exception. It also follows that the connection to the database cannot happen if a CONNECT trigger causes an exception and a transaction cannot start if a TRANSACTION START trigger causes one, either. Both phenomena effectively lock you out of your database until you get in there with database triggers suppressed and fix the bad code.

Suppressing Database Triggers

Some Firebird command-line tools have been supplied with switches that an administrator can use to suppress the automatic firing of database triggers. So far, they are:

```
gbak -nodbtriggers
isql -nodbtriggers
nbackup -T
```

Two-phase Commit

In a two-phase commit scenario, TRANSACTION COMMIT triggers fire in the prepare phase, not at the commit.

Some Caveats

1. The use of the IN AUTONOMOUS TRANSACTION DO statement in the database event triggers related to transactions (TRANSACTION START, TRANSACTION ROLLBACK, TRANSACTION COMMIT) may cause the autonomous transaction to enter an infinite loop

2. The DISCONNECT and TRANSACTION ROLLBACK event triggers will not be executed when clients are disconnected via monitoring tables (DELETE FROM MON$ATTACHMENTS)
Only the database owner and administrators have the authority to create database triggers.

**Examples of CREATE TRIGGER for “Database Triggers”**

1. Creating a trigger for the event of connecting to the database that logs users logging into the system. The trigger is created as inactive.

   ```sql
   CREATE TRIGGER tr_log_connect
   INACTIVE ON CONNECT POSITION 0
   AS
   BEGIN
      INSERT INTO LOG_CONNECT (ID, USERNAME, ATIME)
      VALUES (NEXT VALUE FOR SEQ_LOG_CONNECT, CURRENT_USER, CURRENT_TIMESTAMP);
   END
   ```

2. Creating a trigger for the event of connecting to the database that does not permit any users, except for SYSDBA, to log in during off hours.

   ```sql
   CREATE EXCEPTION E_INCORRECT_WORKTIME 'The working day has not started yet.';
   CREATE TRIGGER TR_LIMIT_WORKTIME ACTIVE ON CONNECT POSITION 1
   AS
   BEGIN
      IF ((CURRENT_USER <> 'SYSDBA') AND
          NOT (CURRENT_TIME BETWEEN time '9:00' AND time '17:00')) THEN
         EXCEPTION E_INCORRECT_WORKTIME;
   END
   ```

**DDL Triggers**

DDL triggers allow restrictions to be placed on users who attempt to create, alter or drop a DDL object. Their other purposes is to keep a metadata change log.

DDL triggers fire on specified metadata changes events in a specified phase. **BEFORE** triggers run before changes to system tables. **AFTER** triggers run after changes in system tables.

⚠️ The event type [BEFORE | AFTER] of a DDL trigger cannot be changed.

In some sense, DDL triggers are a sub-type of database triggers.

**Who Can Create a DDL Trigger?**

DDL triggers can be created by:
• **Administrators**
• **Users with the ALTER DATABASE privilege**

**Suppressing DDL Triggers**

A DDL trigger is a type of database trigger. See **Suppressing Database Triggers** how to suppress database— and DDL— triggers.

**Examples of DDL Triggers**

1. Here is how you might use a DDL trigger to enforce a consistent naming scheme, in this case, stored procedure names should begin with the prefix “SP_”:

```sql
set auto on;
cREATE EXCEPTION e_invalid_sp_name 'Invalid SP name (should start with SP_)';
set term !;
cREATE TRIGGER trig_ddl_sp BEFORE CREATE PROCEDURE
AS
BEGIN
    IF (rdb$get_context('DDL_TRIGGER', 'OBJECT_NAME') NOT STARTING 'SP_') THEN
        EXCEPTION E_INVALID_SP_NAME;
    END;
END;
```

```sql
CREATE PROCEDURE sp_test
AS
BEGIN
END;
```

```sql
CREATE PROCEDURE test
AS
BEGIN
END;
```

-- The last command raises this exception and procedure TEST is not created
-- Statement failed, SQLSTATE = 42000
-- exception 1
-- -E_INVALID_SP_NAME
-- Invalid SP name (should start with SP_)
-- At trigger 'TRIG_DDL_SP' line: 4, col: 5

```sql
set term ;!
```

2. Implement custom DDL security, in this case restricting the running of DDL commands to certain users:
create exception e_access_denied 'Access denied';

set term !;

create trigger trig_ddl before any ddl statement as begin if (current_user <> 'SUPER_USER') then exception e_access_denied; end!

Test

create procedure sp_test as begin end!

-- The last command raises this exception and procedure SP_TEST is not created
-- Statement failed, SQLSTATE = 42000
-- exception 1
-- -E_ACCESS_DENIED
-- -Access denied
-- -At trigger 'TRIG_DDL' line: 4, col: 5

set term ;!

Firebird has privileges for executing DDL statements, so writing a DDL trigger for this should be a last resort, if the same effect cannot be achieved using privileges.

3. Use a trigger to log DDL actions and attempts:

create sequence ddl_seq;

create table ddl_log (id bigint not null primary key, moment timestamp not null, user_name varchar(31) not null, event_type varchar(25) not null, object_type varchar(25) not null, ddl_event varchar(25) not null, object_name varchar(31) not null, sql_text blob sub_type text not null, ok char(1) not null);
The above trigger will fire for this DDL command. It's a good idea to use -nodbtriggers when working with them!
-- (as T1 did not exist, RECREATE acts as CREATE) with OK = Y.
recreate table t1 (  
n1 integer,  
n2 integer  
);  

-- This will fail as T1 already exists, so OK will be N.
create table t1 (  
n1 integer,  
n2 integer  
);  

-- T2 does not exist. There will be no log.
drop table t2;  

-- This will be logged twice  
-- (as T1 exists, RECREATE acts as DROP and CREATE) with OK = Y.
recreate table t1 (  
n integer  
);  

commit;  

select id, ddl_event, object_name, sql_text, ok  
 from ddl_log order by id;

<table>
<thead>
<tr>
<th>ID</th>
<th>DDL_EVENT</th>
<th>OBJECT_NAME</th>
<th>SQL_TEXT</th>
<th>OK</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>CREATE TABLE</td>
<td>T1</td>
<td>recreate table t1 (</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>n1 integer,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>n2 integer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CREATE TABLE</td>
<td>T1</td>
<td>create table t1 (</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>n1 integer,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>n2 integer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>DROP TABLE</td>
<td>T1</td>
<td>recreate table t1 (</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>n integer</td>
<td></td>
</tr>
</tbody>
</table>

Chapter 5. Data Definition (DDL) Statements
See also

ALTER TRIGGER, CREATE OR ALTER TRIGGER, RECREATE TRIGGER, DROP TRIGGER, DDL Triggers in Chapter Procedural SQL (PSQL) Statements

5.7.2. ALTER TRIGGER

Used for

Modifying and deactivating an existing trigger

Available in

DSQL, ESQL

Syntax

ALTER TRIGGER trigname
   [ACTIVE | INACTIVE]
   [{ BEFORE | AFTER } <mutation_list>]
   [ POSITION number ]
   [<module-body>]

!! See syntax of CREATE TRIGGER for further rules !!

The ALTER TRIGGER statement only allows certain changes to the header and body of a trigger.

Permitted Changes to Triggers

- Status (ACTIVE | INACTIVE)
- Phase (BEFORE | AFTER) (of DML triggers)
- Events (of DML triggers)
- Position in the firing order
- Modifications to code in the trigger body

If an element is not specified, it remains unchanged.

A DML trigger cannot be changed to a database (or DDL) trigger.

It is not possible to change the event(s) or phase of a database (or DDL) trigger.
Reminders

The **BEFORE** keyword directs that the trigger be executed before the associated event occurs; the **AFTER** keyword directs that it be executed after the event.

More than one DML event—**INSERT, UPDATE, DELETE**—can be covered in a single trigger. The events should be separated with the keyword **OR**. No event should be mentioned more than once.

The keyword **POSITION** allows an optional execution order (“firing order”) to be specified for a series of triggers that have the same phase and event as their target. The default position is **0**. If no positions are specified, or if several triggers have a single position number, the triggers will be executed in the alphabetical order of their names.

Who Can Alter a Trigger?

DML triggers can be altered by:

- **Administrators**
- The owner of the table (or view)
- Users with the **ALTER ANY TABLE** or— for a view— **ALTER ANY VIEW** privilege

Database and DDL triggers can be altered by:

- **Administrators**
- Users with the **ALTER DATABASE** privilege

Examples using **ALTER TRIGGER**

1. Deactivating the **set_cust_no** trigger (switching it to the inactive status).

   ```sql
   ALTER TRIGGER set_cust_no INACTIVE;
   ```

2. Changing the firing order position of the **set_cust_no** trigger.

   ```sql
   ALTER TRIGGER set_cust_no POSITION 14;
   ```

3. Switching the **TR_CUST_LOG** trigger to the inactive status and modifying the list of events.

   ```sql
   ALTER TRIGGER TR_CUST_LOG
   INACTIVE AFTER INSERT OR UPDATE;
   ```

4. Switching the **tr_log_connect** trigger to the active status, changing its position and body.

   ```sql
   ALTER TRIGGER tr_log_connect
   ```
ACTIVE POSITION 1
AS
BEGIN
    INSERT INTO LOG_CONNECT (ID,
        USERNAME,
        ROLENAME,
        ATIME)
    VALUES (NEXT VALUE FOR SEQ_LOG_CONNECT,
        CURRENT_USER,
        CURRENT_ROLE,
        CURRENT_TIMESTAMP);
END

See also
CREATE TRIGGER, CREATE OR ALTER TRIGGER, RECREATE TRIGGER, DROP TRIGGER

5.7.3. CREATE OR ALTER TRIGGER

Used for
Creating a new trigger or altering an existing trigger

Available in
DSQL

Syntax

CREATE OR ALTER TRIGGER triname
    { <relation_trigger_legacy>
    | <relation_trigger_sql2003>
    | <database_trigger>
    | <ddl_trigger> }
    <module-body>

!! See syntax of CREATE TRIGGER for further rules !!

The CREATE OR ALTER TRIGGER statement creates a new trigger if it does not exist; otherwise it alters and compiles it with the privileges intact and dependencies unaffected.

Example of CREATE OR ALTER TRIGGER

Creating a new trigger if it does not exist or altering it if it does exist

    CREATE OR ALTER TRIGGER set_cust_no
    ACTIVE BEFORE INSERT POSITION 0 ON customer
    AS
    BEGIN
        IF (NEW.cust_no IS NULL) THEN
            NEW.cust_no = GEN_ID(cust_no_gen, 1);
See also
CREATE TRIGGER, ALTER TRIGGER, RECREATE TRIGGER

5.7.4. DROP TRIGGER

Used for
Dropping (deleting) an existing trigger

Available in
DSQL, ESQL

Syntax

```sql
DROP TRIGGER trigname
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trigname</td>
<td>Trigger name</td>
</tr>
</tbody>
</table>

The `DROP TRIGGER` statement drops (deletes) an existing trigger.

**Who Can Drop a Trigger?**

DML triggers can be dropped by:

- **Administrators**
- The owner of the table (or view)
- Users with the `ALTER ANY TABLE` or — for a view — `ALTER ANY VIEW` privilege

Database and DDL triggers can be dropped by:

- **Administrators**
- Users with the `ALTER DATABASE` privilege

**Example of DROP TRIGGER**

*Deleting the set_cust_no trigger*

```sql
DROP TRIGGER set_cust_no;
```

See also
CREATE TRIGGER, RECREATE TRIGGER
5.7.5. RECREATE TRIGGER

Used for
Creating a new trigger or recreating an existing trigger

Available in
DSQL

Syntax

RECREATE TRIGGER triname
   { <relation_trigger_legacy>
     | <relation_trigger_sql2003>
     | <database_trigger>
     | <ddl_trigger> }
<module-body>

!! See syntax of CREATE TRIGGER for further rules !!

The RECREATE TRIGGER statement creates a new trigger if no trigger with the specified name exists; otherwise the RECREATE TRIGGER statement tries to drop the existing trigger and create a new one. The operation will fail on COMMIT if the trigger is in use.

Be aware that dependency errors are not detected until the COMMIT phase of this operation.

Example of RECREATE TRIGGER

Creating or recreating the set_cust_no trigger.

RECREATE TRIGGER set_cust_no
ACTIVE BEFORE INSERT POSITION 0 ON customer
AS
BEGIN
   IF (NEW.cust_no IS NULL) THEN
      NEW.cust_no = GEN_ID(cust_no_gen, 1);
END

See also
CREATE TRIGGER, DROP TRIGGER, CREATE OR ALTER TRIGGER

5.8. PROCEDURE

A stored procedure is a software module that can be called from a client, another procedure, function, executable block or trigger. Stored procedures, stored functions, executable blocks and triggers are written in procedural SQL (PSQL). Most SQL statements are available in PSQL as well, sometimes with some limitations or extensions, notable limitations are DDL and transaction control
Stored procedures can have many input and output parameters.

### 5.8.1. CREATE PROCEDURE

*Used for*
Creating a new stored procedure

*Available in*
DSQL, ESQL

#### Syntax

```plaintext
CREATE PROCEDURE procname [
    ( [ <in_params> ] )
]
[RETURNS (<out_params>)]
<module-body>

<module-body> ::= !! See Syntax of Module Body !!

<in_params> ::= <inparam> [, <inparam> ...]

<inparam> ::= <param_decl> [{= | DEFAULT} <value>]

<out_params> ::= <outparam> [, <outparam> ...]

<outparam> ::= <param_decl>

<value> ::= {<literal> | NULL | <context_var>}

<param_decl> ::= paramname <domain_or_non_array_type> [NOT NULL] [COLLATE collation]

<type> ::= 
    <datatype>
    | [TYPE OF] domain
    | TYPE OF COLUMN rel.col

<domain_or_non_array_type> ::= !! See Scalar Data Types Syntax !!
```

#### Table 42. CREATE PROCEDURE Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>procname</td>
<td>Stored procedure name consisting of up to 31 characters. Must be unique among all table, view and procedure names in the database</td>
</tr>
<tr>
<td>inparam</td>
<td>Input parameter description</td>
</tr>
<tr>
<td>outparam</td>
<td>Output parameter description</td>
</tr>
</tbody>
</table>
The `CREATE PROCEDURE` statement creates a new stored procedure. The name of the procedure must be unique among the names of all stored procedures, tables and views in the database.

`CREATE PROCEDURE` is a *compound statement*, consisting of a header and a body. The header specifies the name of the procedure and declares input parameters and the output parameters, if any, that are to be returned by the procedure.

The procedure body consists of declarations for any local variables and named cursors that will be used by the procedure, followed by one or more statements, or blocks of statements, all enclosed in an outer block that begins with the keyword `BEGIN` and ends with the keyword `END`. Declarations and embedded statements are terminated with semi-colons (`;`).

**Statement Terminators**

Some SQL statement editors—specifically the `isql` utility that comes with Firebird and possibly some third-party editors—employ an internal convention that requires all statements to be terminated with a semi-colon. This creates a conflict with PSQL syntax when coding in these environments. If you are unacquainted with this problem and its solution, please study the details in the PSQL chapter in the section entitled *Switching the Terminator in isql*.

**Parameters**

Each parameter has a data type. The `NOT NULL` constraint can also be specified for any parameter, to prevent `NULL` being passed or assigned to it.

A collation sequence can be specified for string-type parameters, using the `COLLATE` clause.

**Input Parameters**

Input parameters are presented as a parenthesized list following the name of the function. They are passed by value into the procedure, so any changes inside the procedure has no effect on the parameters in the caller. Input parameters may have default values. Parameters with default values specified must be added at the end of the list of parameters.

**Output Parameters**

The optional `RETURNS` clause is for specifying a parenthesised list of output parameters for the stored procedure.
Variable, Cursor and Sub-Routine Declarations

The optional declarations section, located at the start of the body of the procedure definition, defines variables (including cursors) and sub-routines local to the procedure. Local variable declarations follow the same rules as parameters regarding specification of the data type. See details in the PSQL chapter for DECLARE VARIABLE, DECLARE CURSOR, DECLARE FUNCTION, and DECLARE PROCEDURE.

External UDR Procedures

A stored procedure can also be located in an external module. In this case, instead of a procedure body, the CREATE PROCEDURE specifies the location of the procedure in the external module using the EXTERNAL clause. The optional NAME clause specifies the name of the external module, the name of the procedure inside the module, and — optionally — user-defined information. The required ENGINE clause specifies the name of the UDR engine that handles communication between Firebird and the external module. The optional AS clause accepts a string literal “body”, which can be used by the engine or module for various purposes.

Who Can Create a Procedure

The CREATE PROCEDURE statement can be executed by:

• Administrators
• Users with the CREATE PROCEDURE privilege

The user executing the CREATE PROCEDURE statement becomes the owner of the table.

Examples

1. Creating a stored procedure that inserts a record into the BREED table and returns the code of the inserted record:

```sql
CREATE PROCEDURE ADD_BREED (  
  NAME D_BREEDNAME, /* Domain attributes are inherited */  
  NAME_EN TYPE OF D_BREEDNAME, /* Only the domain type is inherited */  
  SHORTNAME TYPE OF COLUMN BREED.SHORTNAME,  
  /* The table column type is inherited */  
  REMARK VARCHAR(120) CHARACTER SET WIN1251 COLLATE PXW_CYRL,  
  CODE_ANIMAL INT NOT NULL DEFAULT 1  
)  
RETURNS (  
  CODE_BREED INT  
)  
AS  
BEGIN  
  INSERT INTO BREED (  
    CODE_ANIMAL, NAME, NAME_EN, SHORTNAME, REMARK)  
  VALUES (  
    :CODE_ANIMAL, :NAME, :NAME_EN, :SHORTNAME, :REMARK)  
  RETURNING CODE_BREED INTO CODE_BREED;
```
CREATE PROCEDURE mail_label (cust_no INTEGER)
RETURNS (line1 CHAR(40), line2 CHAR(40), line3 CHAR(40),
    line4 CHAR(40), line5 CHAR(40), line6 CHAR(40))
AS
    DECLARE VARIABLE customer VARCHAR(25);
    DECLARE VARIABLE first_name VARCHAR(15);
    DECLARE VARIABLE last_name VARCHAR(20);
    DECLARE VARIABLE addr1 VARCHAR(30);
    DECLARE VARIABLE addr2 VARCHAR(30);
    DECLARE VARIABLE city VARCHAR(25);
    DECLARE VARIABLE state VARCHAR(15);
    DECLARE VARIABLE country VARCHAR(15);
    DECLARE VARIABLE postcode VARCHAR(12);
    DECLARE VARIABLE cnt INTEGER;
BEGIN
    line1 = '';
    line2 = '';
    line3 = '';
    line4 = '';
    line5 = '';
    line6 = '';

    SELECT customer, contact_first, contact_last, address_line1, address_line2, city, state_province, country, postal_code
    FROM CUSTOMER
    WHERE cust_no = :cust_no

    IF (customer IS NOT NULL) THEN
        line1 = customer;
    END IF;
    IF (first_name IS NOT NULL) THEN
        line2 = first_name || ' ' || last_name;
    ELSE
        line2 = last_name;
    END IF;
    IF (addr1 IS NOT NULL) THEN
        line3 = addr1;
    END IF;
    IF (addr2 IS NOT NULL) THEN
        line4 = addr2;
    END IF;
    IF (country = 'USA') THEN
        BEGIN
            IF (city IS NOT NULL) THEN
                line5 = city || ', ' || state || ' ' || postcode;
            ELSE
                line5 = '
            END IF;
        END IF;
    ELSE
        line5 = '';
line5 = state || ' ' || postcode;
END
ELSE
BEGIN
  IF (city IS NOT NULL) THEN
    line5 = city || ', ' || state;
  ELSE
    line5 = state;
  END
line6 = country || ' ' || postcode;
END

SUSPEND; -- the statement that sends an output row to the buffer
-- and makes the procedure "selectable"
END

See also
CREATE OR ALTER PROCEDURE, ALTER PROCEDURE, RECREATE PROCEDURE, DROP PROCEDURE

5.8.2. ALTER PROCEDURE

Used for
Modifying an existing stored procedure

Available in
DSQL, ESQL

Syntax

ALTER PROCEDURE procname [ ( [ <in_params> ] ) ]
[RETURNS (<out_params>)]
<module-body>

!! See syntax of CREATE PROCEDURE for further rules !!

The ALTER PROCEDURE statement allows the following changes to a stored procedure definition:

- the set and characteristics of input and output parameters
- local variables
- code in the body of the stored procedure

After ALTER PROCEDURE executes, existing privileges remain intact and dependencies are not affected.

Take care about changing the number and type of input and output parameters in stored procedures. Existing application code and procedures and triggers that call it could become invalid because the new description of the parameters is incompatible with the old calling format. For information on how to troubleshoot such a situation, see the article The RDB$VALID_BLR Field in the Appendix.
Who Can Alter a Procedure

The ALTER PROCEDURE statement can be executed by:

- Administrators
- The owner of the stored procedure
- Users with the ALTER ANY PROCEDURE privilege

ALTER PROCEDURE Example

Altering the GET_EMP_PROJ stored procedure.

```sql
ALTER PROCEDURE GET_EMP_PROJ (
    EMP_NO SMALLINT
) RETURNS (
    PROJ_ID VARCHAR(20)
) AS
BEGIN
    FOR SELECT
        PROJ_ID
    FROM
        EMPLOYEE_PROJECT
    WHERE
        EMP_NO = :emp_no
    INTO :proj_id
    DO
    SUSPEND;
END
```

See also

CREATE PROCEDURE, CREATE OR ALTER PROCEDURE, RECREATE PROCEDURE, DROP PROCEDURE

5.8.3. CREATE OR ALTER PROCEDURE

Used for
Creating a new stored procedure or altering an existing one

Available in
DSQL

Syntax

```sql
CREATE OR ALTER PROCEDURE procname [ ( [ <in_params> ] ) ]
[RETURNS (<out_params>)]
<module-body>

!! See syntax of CREATE PROCEDURE for further rules !!
```
The `CREATE OR ALTER PROCEDURE` statement creates a new stored procedure or alters an existing one. If the stored procedure does not exist, it will be created by invoking a `CREATE PROCEDURE` statement transparently. If the procedure already exists, it will be altered and compiled without affecting its existing privileges and dependencies.

**CREATE OR ALTER PROCEDURE Example**

Creating or altering the `GET_EMP_PROJ` procedure.

```sql
CREATE OR ALTER PROCEDURE GET_EMP_PROJ (
    EMP_NO SMALLINT)
RETURNS (
    PROJ_ID VARCHAR(20))
AS
BEGIN
    FOR SELECT
        PROJ_ID
    FROM
        EMPLOYEE_PROJECT
    WHERE
        EMP_NO = :emp_no
    INTO :proj_id
    DO
        SUSPEND;
    END
```

*See also*

`CREATE PROCEDURE, ALTER PROCEDURE, RECREATE PROCEDURE`

### 5.8.4. DROP PROCEDURE

**Used for**

Deleting a stored procedure

**Available in**

DSQL, ESQL

**Syntax**

```sql
DROP PROCEDURE procname
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>procname</td>
<td>Name of an existing stored procedure</td>
</tr>
</tbody>
</table>

The `DROP PROCEDURE` statement deletes an existing stored procedure. If the stored procedure has any dependencies, the attempt to delete it will fail and the appropriate error will be raised.
Who Can Drop a Procedure

The ALTER PROCEDURE statement can be executed by:

- Administrators
- The owner of the stored procedure
- Users with the DROP ANY PROCEDURE privilege

DROP PROCEDURE Example

Deleting the GET_EMP_PROJ stored procedure.

```
DROP PROCEDURE GET_EMP_PROJ;
```

See also

CREATE PROCEDURE, RECREATE PROCEDURE

5.8.5. RECREATE PROCEDURE

Used for
Creating a new stored procedure or recreating an existing one

Available in
DSQL

Syntax

```
RECREATE PROCEDURE procname [ ( [ <in_params> ] ) ]
[RETURNS (<out_params>)]
<module-body>

!! See syntax of CREATE PROCEDURE for further rules !!
```

The RECREATE PROCEDURE statement creates a new stored procedure or recreates an existing one. If there is a procedure with this name already, the engine will try to delete it and create a new one. Recreating an existing procedure will fail at the COMMIT request if the procedure has dependencies.

⚠️ Be aware that dependency errors are not detected until the COMMIT phase of this operation.

After a procedure is successfully recreated, privileges to execute the stored procedure, and the privileges of the stored procedure itself are dropped.

RECREATE PROCEDURE Example

Creating the new GET_EMP_PROJ stored procedure or recreating the existing GET_EMP_PROJ stored procedure.

```
RECREATE PROCEDURE GET_EMP_PROJ ( 
```
CREATE FUNCTION funcname [ ( [ <in_params> ] ) ]
RETURNS <domain_or_non_array_type> [COLLATE collation]
[DETERMINISTIC]
<module-body>

<!-- See Syntax of Module Body -->
CREATE FUNCTION is a compound statement with a header and a body. The header defines the name of the stored function, and declares input parameters and return type.

The function body consists of optional declarations of local variables, named cursors, and subroutines (sub-functions and sub-procedures), and one or more statements or statement blocks, enclosed in an outer block that starts with the keyword BEGIN and ends with the keyword END. Declarations and statements inside the function body must be terminated with a semicolon (‘;’).
Statement Terminators

Some SQL statement editors—specifically the isql utility that comes with Firebird and possibly some third-party editors—employ an internal convention that requires all statements to be terminated with a semi-colon. This creates a conflict with PSQL syntax when coding in these environments. If you are unacquainted with this problem and its solution, please study the details in the PSQL chapter in the section entitled Switching the Terminator in isql.

Parameters

Each parameter has a data type.

A collation sequence can be specified for string-type parameters, using the COLLATE clause.

Input Parameters

Input parameters are presented as a parenthesized list following the name of the function. They are passed by value into the function, so any changes inside the function has no effect on the parameters in the caller. The NOT NULL constraint can also be specified for any input parameter, to prevent NULL being passed or assigned to it. Input parameters may have default values. Parameters with default values specified must be added at the end of the list of parameters.

Output Parameter

The RETURNS clause specifies the return type of the stored function. If a function returns a string value, then it is possible to specify the collation using the COLLATE clause. As a return type, you can specify a data type, a domain name, the type of a domain (using TYPE OF), or the type of a column of a table or view (using TYPE OF COLUMN).

Deterministic functions

The optional DETERMINISTIC clause indicates that the function is deterministic. Deterministic functions always return the same result for the same set of inputs. Non-deterministic functions can return different results for each invocation, even for the same set of inputs. If a function is specified as deterministic, then such a function might not be called again if it has already been called once with the given set of inputs, and instead takes the result from a metadata cache.

Current versions of Firebird do not actually cache results of deterministic functions.

Specifying the DETERMINISTIC clause is actually something like a “promise” that the function will return the same thing for equal inputs. At the moment, a deterministic function is considered an invariant, and works like other invariants. That is, they are computed and cached at the current execution level of a given statement.

This is easily demonstrated with an example:

```sql
CREATE FUNCTION FN_T
RETURNS DOUBLE PRECISION DETERMINISTIC
AS
```
BEGIN
    RETURN rand();
END;

-- the function will be evaluated twice and will return 2 different values
SELECT fn_t() FROM rdb$database
UNION ALL
SELECT fn_t() FROM rdb$database;

-- the function will be evaluated once and will return 2 identical values
WITH t (n) AS (
    SELECT 1 FROM rdb$database
    UNION ALL
    SELECT 2 FROM rdb$database
)
SELECT n, fn_t() FROM t;

Variable, Cursor and Sub-Routine Declarations

The optional declarations section, located at the start of the body of the function definition, defines variables (including cursors) and sub-routines local to the function. Local variable declarations follow the same rules as parameters regarding specification of the data type. See details in the PSQL chapter for DECLARE VARIABLE, DECLARE CURSOR, DECLARE FUNCTION, and DECLARE PROCEDURE.

Function Body

The header section is followed by the function body, consisting of one or more PSQL statements enclosed between the outer keywords BEGIN and END. Multiple BEGIN … END blocks of terminated statements may be embedded inside the procedure body.

External UDR Functions

A stored function can also be located in an external module. In this case, instead of a function body, the CREATE FUNCTION specifies the location of the function in the external module using the EXTERNAL clause. The optional NAME clause specifies the name of the external module, the name of the function inside the module, and — optionally — user-defined information. The required ENGINE clause specifies the name of the UDR engine that handles communication between Firebird and the external module. The optional AS clause accepts a string literal “body”, which can be used by the engine or module for various purposes.

External UDR (User Defined Routine) functions created using CREATE FUNCTION … EXTERNAL … should not be confused with legacy UDFs (User Defined Functions) declared using DECLARE EXTERNAL FUNCTION.

UDFs are deprecated, and a legacy from previous Firebird functions. Their capabilities are significantly inferior to the capabilities to the new type of external UDR functions.
Who Can Create a Function

The CREATE FUNCTION statement can be executed by:

- Administrators
- Users with the CREATE FUNCTION privilege

The user who created the stored function becomes its owner.

CREATE FUNCTION Examples

1. Creating a stored function

```sql
CREATE FUNCTION ADD_INT (A INT, B INT DEFAULT 0)
RETURNS INT
AS
BEGIN
    RETURN A + B;
END
```

Calling in a select:

```sql
SELECT ADD_INT(2, 3) AS R FROM RDB$DATABASE
```

Call inside PSQL code, the second optional parameter is not specified:

```sql
MY_VAR = ADD_INT(A);
```

2. Creating a deterministic stored function

```sql
CREATE FUNCTION FN_E()
RETURNS DOUBLE PRECISION DETERMINISTIC
AS
BEGIN
    RETURN EXP(1);
END
```

3. Creating a stored function with table column type parameters

Returns the name of a type by field name and value

```sql
CREATE FUNCTION GET_MNEMONIC (
    AFIELD_NAME TYPE OF COLUMN RDB$TYPES.RDB$FIELD_NAME,
    ATYPE TYPE OF COLUMN RDB$TYPES.RDB$TYPE)
RETURNS TYPE OF COLUMN RDB$TYPES.RDB$TYPE_NAME
AS
```
BEGIN
    RETURN (SELECT RDB$TYPE_NAME
            FROM RDB$TYPES
            WHERE RDB$FIELD_NAME = :AFIELD_NAME
            AND RDB$TYPE = :ATYPE);
END

4. Creating an external stored function

Create a function located in an external module (UDR). Function implementation is located in the external module `udrcpp_example`. The name of the function inside the module is `wait_event`.

```sql
CREATE FUNCTION wait_event (event_name varchar (31) CHARACTER SET ascii )
    RETURNS INTEGER
    EXTERNAL NAME 'udrcpp_example!Wait_event'
    ENGINE udr
```

5. Creating a stored function containing a sub-function

Creating a function to convert a number to hexadecimal format.

```sql
CREATE FUNCTION INT_TO_HEX (ANumber BIGINT , AByte_Per_Number SMALLINT = 8)
    RETURNS CHAR (66)
AS
    DECLARE VARIABLE xMod SMALLINT ;
    DECLARE VARIABLE xResult VARCHAR (64);
    DECLARE FUNCTION TO_HEX (ANum SMALLINT ) RETURNS CHAR
        AS
        BEGIN
            RETURN CASE ANum
            WHEN 0 THEN '0'
            WHEN 1 THEN '1'
            WHEN 2 THEN '2'
            WHEN 3 THEN '3'
            WHEN 4 THEN '4'
            WHEN 5 THEN '5'
            WHEN 6 THEN '6'
            WHEN 7 THEN '7'
            WHEN 8 THEN '8'
            WHEN 9 THEN '9'
            WHEN 10 THEN 'A'
            WHEN 11 THEN 'B'
            WHEN 12 THEN 'C'
            WHEN 13 THEN 'D'
            WHEN 14 THEN 'E'
            WHEN 15 THEN 'F'
            ```
ELSE NULL
END;
END
BEGIN
  xMod = MOD (ANumber, 16);
  ANumber = ANumber / 16;
  xResult = TO_HEX (xMod);
  WHILE (ANUMBER> 0) DO
    BEGIN
      xMod = MOD (ANumber, 16);
      ANumber = ANumber / 16;
      xResult = TO_HEX (xMod) || xResult;
    END
    RETURN '0x' || LPAD (xResult, AByte_Per_Number * 2, '0 ');
  END
END

See also

CREATE OR ALTER FUNCTION, ALTER FUNCTION, RECREATE FUNCTION, DROP FUNCTION, DECLARE EXTERNAL FUNCTION

5.9.2. ALTER FUNCTION

Used for

Modifying an existing stored function

Available in

DSQL

Syntax

ALTER FUNCTION funcname
  [ ( [ <in_params> ] ) ]
  RETURNS <domain_or_non_array_type> [COLLATE collation]
  [DETERMINISTIC]
  <module-body>

!! See syntax of CREATE FUNCTION for further rules !!

The ALTER FUNCTION statement allows the following changes to a stored function definition:

- the set and characteristics of input and output type
- local variables, named cursors, and sub-routines
- code in the body of the stored procedure

For external functions (UDR), you can change the entry point and engine name. For legacy external functions declared using DECLARE EXTERNAL FUNCTION — also known as UDFs — it is not possible to convert to PSQL and vice versa.
After **ALTER FUNCTION** executes, existing privileges remain intact and dependencies are not affected. Take care about changing the number and type of input parameters and the output type of a stored function. Existing application code and procedures, functions and triggers that call it could become invalid because the new description of the parameters is incompatible with the old calling format. For information on how to troubleshoot such a situation, see the article **The RDB$VALID_BLR Field** in the Appendix.

**Who Can Alter a Function**

The **ALTER FUNCTION** statement can be executed by:

- Administrators
- Owner of the stored function
- Users with the **ALTER ANY FUNCTION** privilege

**Examples of **ALTER FUNCTION**

**Altering a stored function**

```sql
ALTER FUNCTION ADD_INT(A INT, B INT, C INT)
RETURNS INT
AS
BEGIN
   RETURN A + B + C;
END
```

**See also**

**CREATE FUNCTION, CREATE OR ALTER FUNCTION, RECREATE FUNCTION, DROP FUNCTION**

### 5.9.3. CREATE OR ALTER FUNCTION

**Used for**

Creating a new or modifying an existing stored function

**Available in**

DSQL

**Syntax**

```sql
CREATE OR ALTER FUNCTION funcname
   [ ( [ <in_params> ] ) ]
RETURNS <domain_or_non_array_type> [COLLATE collation]
[DETERMINISTIC]
<module-body>
```
The **CREATE OR ALTER FUNCTION** statement creates a new stored function or alters an existing one. If the stored function does not exist, it will be created by invoking a **CREATE FUNCTION** statement transparently. If the function already exists, it will be altered and compiled (through **ALTER FUNCTION**) without affecting its existing privileges and dependencies.

### Examples of **CREATE OR ALTER FUNCTION**

Create a new or alter an existing stored function

```sql
CREATE OR ALTER FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT
AS
BEGIN
    RETURN A + B;
END
```

See also

**CREATE FUNCTION**, **ALTER FUNCTION**, **DROP FUNCTION**

### 5.9.4. **DROP FUNCTION**

**Used for**

Dropping a stored function

**Available in**

DSQL

**Syntax**

```
DROP FUNCTION funcname
```

**Table 45. **DROP FUNCTION** Statement Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>funcname</td>
<td>Stored function name consisting of up to 31 characters. Must be unique among all function names in the database.</td>
</tr>
</tbody>
</table>

The **DROP FUNCTION** statement deletes an existing stored function. If the stored function has any dependencies, the attempt to delete it will fail and the appropriate error will be raised.

### Who Can Drop a Function

The **DROP FUNCTION** statement can be executed by:

- Administrators
- Owner of the stored function
• Users with the DROP ANY FUNCTION privilege

**Examples of DROP FUNCTION**

```
DROP FUNCTION ADD_INT;
```

**See also**

CREATE FUNCTION, CREATE OR ALTER FUNCTION, RECREATE FUNCTION

### 5.9.5. RECREATE FUNCTION

**Used for**

Creating a new stored function or recreating an existing one

**Available in**

DSQL

**Syntax**

```
RECREATE FUNCTION funcname
[ ( [ <in_params> ] ) ]
RETURNS <domain_or_non_array_type> [COLLATE collation]
[DETERMINISTIC]
<module-body>
```

!! See syntax of CREATE FUNCTION for further rules !!

The **RECREATE FUNCTION** statement creates a new stored function or recreates an existing one. If there is a function with this name already, the engine will try to drop it and then create a new one. Recreating an existing function will fail at **COMMIT** if the function has dependencies.

Be aware that dependency errors are not detected until the **COMMIT** phase of this operation.

After a procedure is successfully recreated, existing privileges to execute the stored function and the privileges of the stored function itself are dropped.

**Examples of RECREATE FUNCTION**

**Creating or recreating a stored function**

```
RECREATE FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT
AS
BEGIN
    RETURN A + B;
END
```
See also

CREATE FUNCTION, DROP FUNCTION

5.10. EXTERNAL FUNCTION

External functions, also known as “User-Defined Functions” (UDFs) are programs written in an external programming language and stored in dynamically loaded libraries. Once declared in a database, they become available in dynamic and procedural statements as though they were implemented in the SQL language.

External functions extend the possibilities for processing data with SQL considerably. To make a function available to a database, it is declared using the statement DECLARE EXTERNAL FUNCTION.

The library containing a function is loaded when any function included in it is called.

External functions declared as DECLARE EXTERNAL FUNCTION are a legacy from previous versions of Firebird. Their capabilities are inferior to the capabilities of the new type of external functions, UDR (User-Defined Routine). Such functions are declared as CREATE FUNCTION … EXTERNAL …. See CREATE FUNCTION for details.

External functions may be contained in more than one library — or “module”, as it is referred to in the syntax.

UDFs are fundamentally insecure. We recommend avoiding their use whenever possible, and disabling UDFs in your database configuration (UdfAccess = None in firebird.conf). If you do need to call native code from your database, use a UDR external engine instead.

See also

FUNCTION

5.10.1. DECLARE EXTERNAL FUNCTION

Used for
Declaring a user-defined function (UDF) to the database

Available in
DSQL, ESQL

Syntax

```
DECLARE EXTERNAL FUNCTION funcname
   [ [ <arg_desc_list> | ( <arg_desc_list> ) ] ]
   RETURNS { <return_value> | ( <return_value> ) }
   ENTRY_POINT 'entry_point' MODULE_NAME 'library_name'
```

<arg_desc_list> ::= 
   <arg_type_decl> [ , <arg_type_decl> ... ]
<arg_type_decl> ::= 
  <udf_data_type> [BY {DESCRIPTOR | SCALAR_ARRAY} | NULL]

<udf_data_type> ::= 
  <scalar_datatype> | BLOB | CSTRING(length) [ CHARACTER SET charset ]

<scalar_datatype> ::= 
  !! See Scalar Data Types Syntax !!

<return_value> ::= 
  { <udf_data_type> | PARAMETER param_num } 
  [[{ BY VALUE | BY DESCRIPTOR [FREE_IT] | FREE_IT }]

Table 46. DECLARE EXTERNAL FUNCTION Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>funcname</td>
<td>Function name in the database. It may consist of up to 31 characters. It should be unique among all internal and external function names in the database and need not be the same name as the name exported from the UDF library via ENTRY_POINT.</td>
</tr>
<tr>
<td>entry_point</td>
<td>The exported name of the function</td>
</tr>
<tr>
<td>library_name</td>
<td>The name of the module (MODULE_NAME) from which the function is exported. This will be the name of the file, without the “.dll” or “.so” file extension.</td>
</tr>
<tr>
<td>length</td>
<td>The maximum length of a null-terminated string, specified in bytes</td>
</tr>
<tr>
<td>charset</td>
<td>Character set of the CSTRING</td>
</tr>
<tr>
<td>param_num</td>
<td>The number of the input parameter, numbered from 1 in the list of input parameters in the declaration, describing the data type that will be returned by the function</td>
</tr>
</tbody>
</table>

The DECLARE EXTERNAL FUNCTION statement makes a user-defined function available in the database. UDF declarations must be made in each database that is going to use them. There is no need to declare UDFs that will never be used.

The name of the external function must be unique among all function names. It may be different from the exported name of the function, as specified in the ENTRY_POINT argument.

**DECLARE EXTERNAL FUNCTION Input Parameters**

The input parameters of the function follow the name of the function and are separated with commas. Each parameter has an SQL data type specified for it. Arrays cannot be used as function parameters. In addition to the SQL types, the CSTRING type is available for specifying a null-terminated string with a maximum length of LENGTH bytes. There are several mechanisms for passing a parameter from the Firebird engine to an external function, each of these mechanisms
will be discussed below.

By default, input parameters are passed by reference. There is no separate clause to explicitly indicate that parameters are passed by reference.

When passing a NULL value by reference, it is converted to the equivalent of zero, for example, a number ‘0’ or an empty string (“’”). If the keyword NULL is specified after a parameter, then with passing a NULL values, the null pointer will be passed to the external function.

---

Declaring a function with the NULL keyword does not guarantee that the function will correctly handle a NULL input parameter. Any function must be written or rewritten to correctly handle NULL values. Always use the function declaration as provided by its developer.

---

If BY DESCRIPTOR is specified, then the input parameter is passed by descriptor. In this case, the UDF parameter will receive a pointer to an internal structure known as a descriptor. The descriptor contains information about the datatype, subtype, precision, character set and collation, scale, a pointer to the data itself and some flags, including the NULL indicator. This declaration only works if the external function is written using a handle.

---

When passing a function parameter by descriptor, the passed value is not cast to the declared data type.

---

The BY SCALAR_ARRAY clause is used when passing arrays as input parameters. Unlike other types, you cannot return an array from a UDF.

---

**Clauses and Keywords**

**RETURNS clause**

(Required) specifies the output parameter returned by the function. A function is scalar, it returns one value (output parameter). The output parameter can be of any SQL type (except an array or an array element) or a null-terminated string (CSTRING). The output parameter can be passed by reference (the default), by descriptor or by value. If the BY DESCRIPTOR clause is specified, the output parameter is passed by descriptor. If the BY VALUE clause is specified, the output parameter is passed by value.

**PARAMETER keyword**

specifies that the function returns the value from the parameter under number param_num. It is necessary if you need to return a value of data type BLOB.

**FREE_IT keyword**

means that the memory allocated for storing the return value will be freed after the function is executed. It is used only if the memory was allocated dynamically in the UDF. In such a UDF, the memory must be allocated with the help of the ib_util_malloc function from the ib_util module, a requirement for compatibility with the functions used in Firebird code and in the code of the shipped UDF modules, for allocating and freeing memory.
ENTRY_POINT clause

specifies the name of the entry point (the name of the imported function), as exported from the module.

MODULE_NAME clause

defines the name of the module where the exported function is located. The link to the module should not be the full path and extension of the file, if that can be avoided. If the module is located in the default location (in the ../UDF subdirectory of the Firebird server root) or in a location explicitly configured in firebird.conf, it makes it easier to move the database between different platforms. The UDFAccess parameter in the firebird.conf file allows access restrictions to external functions modules to be configured.

Any user connected to the database can declare an external function (UDF).

Who Can Create an External Function

The DECLARE EXTERNAL FUNCTION statement can be executed by:

- Administrators
- Users with the CREATE FUNCTION privilege

The user who created the function becomes its owner.

Examples using DECLARE EXTERNAL FUNCTION

1. Declaring the addDay external function located in the fbudf module. The input and output parameters are passed by reference.

   DECLARE EXTERNAL FUNCTION addDay
   TIMESTAMP, INT
   RETURNS TIMESTAMP
   ENTRY_POINT 'addDay' MODULE_NAME 'fbudf';

2. Declaring the invl external function located in the fbudf module. The input and output parameters are passed by descriptor.

   DECLARE EXTERNAL FUNCTION invl
   INT BY DESCRIPTOR, INT BY DESCRIPTOR
   RETURNS INT BY DESCRIPTOR
   ENTRY_POINT 'idNvl' MODULE_NAME 'fbudf';

3. Declaring the isLeapYear external function located in the fbudf module. The input parameter is passed by reference, while the output parameter is passed by value.

   DECLARE EXTERNAL FUNCTION isLeapYear
   TIMESTAMP
   RETURNS INT BY VALUE
ENTRY_POINT 'isLeapYear' MODULE_NAME 'fbudf';

4. Declaring the i64Truncate external function located in the fbudf module. The input and output parameters are passed by descriptor. The second parameter of the function is used as the return value.

DECLARE EXTERNAL FUNCTION i64Truncate
    NUMERIC(18) BY DESCRIPTOR, NUMERIC(18) BY DESCRIPTOR
    RETURNS PARAMETER 2
    ENTRY_POINT 'fbtruncate' MODULE_NAME 'fbudf';

See also
ALTER EXTERNAL FUNCTION, DROP EXTERNAL FUNCTION, CREATE FUNCTION

5.10.2. ALTER EXTERNAL FUNCTION

Used for
Changing the entry point and/or the module name for a user-defined function (UDF)

Available in
DSQL

Syntax

```
ALTER EXTERNAL FUNCTION funcname
    [ENTRY_POINT 'new_entry_point']
    [MODULE_NAME 'new_library_name']
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>funcname</td>
<td>Function name in the database</td>
</tr>
<tr>
<td>new_entry_point</td>
<td>The new exported name of the function</td>
</tr>
<tr>
<td>new_library_name</td>
<td>The new name of the module (MODULE_NAME from which the function is exported). This will be the name of the file, without the “.dll” or “.so” file extension.</td>
</tr>
</tbody>
</table>

The ALTER EXTERNAL FUNCTION statement changes the entry point and/or the module name for a user-defined function (UDF). Existing dependencies remain intact after the statement containing the change(s) is executed.

The ENTRY_POINT clause
    is for specifying the new entry point (the name of the function as exported from the module).

The MODULE_NAME clause
    is for specifying the new name of the module where the exported function is located.
Any user connected to the database can change the entry point and the module name.

**Who Can Alter an External Function**

The `ALTER EXTERNAL FUNCTION` statement can be executed by:

- Administrators
- Owner of the external function
- Users with the `ALTER ANY FUNCTION` privilege

**Examples using ALTER EXTERNAL FUNCTION**

*Changing the entry point for an external function*

```sql
ALTER EXTERNAL FUNCTION invl ENTRY_POINT 'intNvl';
```

*Changing the module name for an external function*

```sql
ALTER EXTERNAL FUNCTION invl MODULE_NAME 'fbudf2';
```

See also

`DECLARE EXTERNAL FUNCTION, DROP EXTERNAL FUNCTION`

### 5.10.3. DROP EXTERNAL FUNCTION

**Used for**

Removing a user-defined function (UDF) from a database

**Available in**

DSQL, ESQL

**Syntax**

```sql
DROP EXTERNAL FUNCTION funcname
```

**Table 48. DROP EXTERNAL FUNCTION Statement Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>funcname</td>
<td>Function name in the database</td>
</tr>
</tbody>
</table>

The `DROP EXTERNAL FUNCTION` statement deletes the declaration of a user-defined function from the database. If there are any dependencies on the external function, the statement will fail and the appropriate error will be raised.

Any user connected to the database can delete the declaration of an internal function.
Who Can Drop an External Function

The DROP EXTERNAL FUNCTION statement can be executed by:

- Administrators
- Owner of the external function
- Users with the DROP ANY FUNCTION privilege

Example using DROP EXTERNAL FUNCTION

Deleting the declaration of the addDay function.

DROP EXTERNAL FUNCTION addDay;

See also

DECLARE EXTERNAL FUNCTION

5.11. PACKAGE

A package is a group of procedures and functions managed as one entity.

5.11.1. CREATE PACKAGE

Used for
Declaring the package header

Available in
DSQL

Syntax

CREATE PACKAGE package_name
AS
BEGIN
[ <package_item> ... ]
END

<package_item> ::= 
  <function_decl>;
  | <procedure_decl>;

<function_decl> ::= 
  FUNCTION funcname [ ( [ <in_params> ] ) ]
  RETURNS <domain_or_non_array_type> [COLLATE collation]
  [DETERMINISTIC]

<procedure_decl> ::= 
  PROCEDURE procname [ ( [ <in_params> ] ) ]
  [RETURNS (<out_params>)]
<in_params> ::= <inparam> [, <inparam> ... ]

<inparam> ::= <param_decl> [ { = | DEFAULT } <value> ]

<out_params> ::= <outparam> [, <outparam> ... ]

<outparam> ::= <param_decl>

<value> ::= { literal | NULL | context_var }

<param_decl> ::= paramname <domain_or_non_array_type> [NOT NULL]
[COLLATE collation]

<domain_or_non_array_type> ::= !! See Scalar Data Types Syntax !!

Table 49. CREATE PACKAGE Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>package_name</td>
<td>Package name consisting of up to 31 characters. The package name must be unique among all package names.</td>
</tr>
<tr>
<td>function_decl</td>
<td>Function declaration</td>
</tr>
<tr>
<td>procedure_decl</td>
<td>Procedure declaration</td>
</tr>
<tr>
<td>func_name</td>
<td>Function name consisting of up to 31 characters. The function name must be unique within the package.</td>
</tr>
<tr>
<td>proc_name</td>
<td>Procedure name consisting of up to 31 characters. The function name must be unique within the package.</td>
</tr>
<tr>
<td>collation</td>
<td>Collation sequence</td>
</tr>
<tr>
<td>inparam</td>
<td>Input parameter declaration</td>
</tr>
<tr>
<td>outparam</td>
<td>Output parameter declaration</td>
</tr>
<tr>
<td>literal</td>
<td>A literal value that is assignment-compatible with the data type of the parameter</td>
</tr>
<tr>
<td>context_var</td>
<td>Any context variable that is assignment-compatible with the data type of the parameter</td>
</tr>
<tr>
<td>paramname</td>
<td>The name of an input parameter of a procedure or function, or an output parameter of a procedure. It may consist of up to 31 characters. The name of the parameter must be unique among input and output parameters of the procedure or function.</td>
</tr>
</tbody>
</table>

The CREATE PACKAGE statement creates a new package header. Routines (procedures and functions) declared in the package header are available outside the package using the full identifier (package_name.proc_name or package_name.func_name). Routines defined only in the package body — but not in the package header — are not visible outside the package.
Package procedure and function names may shadow global routines

If a package header or package body declares a procedure or function with the same name as a stored procedure or function in the global namespace, it is not possible to call that global procedure or function from the package body. In this case, the procedure or function of the package will always be called.

For this reason, it is recommended that the names of stored procedures and functions in packages do not overlap with names of stored procedures and functions in the global namespace.

Statement Terminators

Some SQL statement editors—specifically the *isql* utility that comes with Firebird and possibly some third-party editors—employ an internal convention that requires all statements to be terminated with a semicolon. This creates a conflict with PSQL syntax when coding in these environments. If you are unacquainted with this problem and its solution, please study the details in the PSQL chapter in the section entitled Switching the Terminator in *isql*.

Procedure and Function Parameters

For details on stored procedure parameters, see Parameters in CREATE PROCEDURE.

For details on function parameters, see Parameters in CREATE FUNCTION.

Who Can Create a Package

The CREATE PACKAGE statement can be executed by:

- Administrators
- Users with the CREATE PACKAGE privilege

The user who created the package header becomes its owner.

Examples of CREATE PACKAGE

Create a package header

```sql
CREATE PACKAGE APP_VAR
AS
BEGIN
  FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
  FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
  PROCEDURE SET_DATERANGE(ADATEBEGIN DATE,
                            ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

See also

CREATE PACKAGE BODY, RECREATE PACKAGE BODY, ALTER PACKAGE, DROP PACKAGE, RECREATE PACKAGE
5.11.2. ALTER PACKAGE

Used for
Altering the package header

Available in
DSQL

Syntax

```
ALTER PACKAGE package_name
AS
BEGIN
  [ <package_item> ... ]
END
```

!! See syntax of CREATE PACKAGE for further rules!!

The ALTER PACKAGE statement modifies the package header. It can be used to change the number and definition of procedures and functions, including their input and output parameters. However, the source and compiled form of the package body is retained, though the body might be incompatible after the change to the package header. The validity of a package body for the defined header is stored in the column RDB$PACKAGES.RDB$VALID_BODY_FLAG.

Who Can Alter a Package

The ALTER PACKAGE statement can be executed by:

- Administrators
- The owner of the package
- Users with the ALTER ANY PACKAGE privilege

Examples of ALTER PACKAGE

Modifying a package header

```
ALTER PACKAGE APP_VAR
AS
BEGIN
  FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
  FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
  PROCEDURE SET_DATERANGE(ADATEBEGIN DATE,
                           ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

See also

CREATE PACKAGE, DROP PACKAGE, RECREATE PACKAGE BODY
5.11.3. CREATE OR ALTER PACKAGE

Used for
Creating a new or altering an existing package header

Available in
DSQL

Syntax

```sql
CREATE OR ALTER PACKAGE package_name
AS
BEGIN
   [ <package_item> ... ]
END
```

!! See syntax of CREATE PACKAGE for further rules!!

The CREATE OR ALTER PACKAGE statement creates a new package or modifies an existing package header. If the package header does not exist, it will be created using CREATE PACKAGE. If it already exists, then it will be modified using ALTER PACKAGE while retaining existing privileges and dependencies.

Examples of CREATE OR ALTER PACKAGE

Creating a new or modifying an existing package header

```sql
CREATE OR ALTER PACKAGE APP_VAR
AS
BEGIN
   FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
   FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
   PROCEDURE SET_DATERANGE(ADATEBEGIN DATE,
                              ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

See also
CREATE PACKAGE, ALTER PACKAGE, RECREATE PACKAGE, RECREATE PACKAGE BODY

5.11.4. DROP PACKAGE

Used for
Dropping a package header

Available in
DSQL
Syntax

```sql
DROP PACKAGE package_name
```

### Table 50. DROP PACKAGE Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>package_name</td>
<td>Package name</td>
</tr>
</tbody>
</table>

The `DROP PACKAGE` statement deletes an existing package header. If a package body exists, it will be dropped together with the package header. If there are still dependencies on the package, an error will be raised.

#### Who Can Drop a Package

The `DROP PACKAGE` statement can be executed by:

- Administrators
- The owner of the package
- Users with the `DROP ANY PACKAGE` privilege

#### Examples of DROP PACKAGE

**Dropping a package header**

```sql
DROP PACKAGE APP_VAR
```

See also

`CREATE PACKAGE, DROP PACKAGE BODY`

### 5.11.5. RECREATE PACKAGE

**Used for**

Creating a new or recreating an existing package header

**Available in**

DSQL

**Syntax**

```sql
RECREATE PACKAGE package_name
AS
BEGIN
  [ <package_item> ... ]
END
```

!! See syntax of `CREATE PACKAGE` for further rules!!
The `RECREATE PACKAGE` statement creates a new package or recreates an existing package header. If a package header with the same name already exists, then this statement will first drop it and then create a new package header. It is not possible to recreate the package header if there are still dependencies on the existing package, or if the body of the package exists. Existing privileges of the package itself are not preserved, nor are privileges to execute the procedures or functions of the package.

**Examples of RECREATE PACKAGE**

Creation a new or recreating an existing package header

```
RECREATE PACKAGE APP_VAR AS
BEGIN
  FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
  FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
  PROCEDURE SET_DATERANGE(ADATEBEGIN DATE,
                           ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

See also

CREATE PACKAGE, DROP PACKAGE, CREATE PACKAGE BODY, RECREATE PACKAGE BODY

### 5.12. PACKAGE BODY

#### 5.12.1. CREATE PACKAGE BODY

*Used for*

Creating the package body

*Available in*

DSQL

*Syntax*

```
CREATE PACKAGE BODY name AS
BEGIN
  [ <package_item> ... ]
  [ <package_body_item> ... ]
END

<package_item> ::= !! See CREATE PACKAGE syntax !!

<package_body_item> ::= 
  <function_impl> | 
  <procedure_impl>
```
Table 51. CREATE PACKAGE BODY Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>package_name</td>
<td>Package name consisting of up to 31 characters. The package name must be unique among all package names.</td>
</tr>
<tr>
<td>function_impl</td>
<td>Function implementation. Essentially a CREATE FUNCTION statement without CREATE.</td>
</tr>
<tr>
<td>procedure_impl</td>
<td>Procedure implementation Essentially a CREATE PROCEDURE statement without CREATE.</td>
</tr>
<tr>
<td>func_name</td>
<td>Function name consisting of up to 31 characters. The function name must be unique within the package.</td>
</tr>
<tr>
<td>collation</td>
<td>Collation sequence</td>
</tr>
<tr>
<td>proc_name</td>
<td>Procedure name consisting of up to 31 characters. The function name must be unique within the package.</td>
</tr>
</tbody>
</table>

The CREATE PACKAGE BODY statement creates a new package body. The package body can only be created after the package header has been created. If there is no package header with name package_name, an appropriate error will be raised.

All procedures and functions declared in the package header must be implemented in the package body. Additional procedures and functions may be defined and implemented in the package body only. Procedure and functions defined in the package body, but not defined in the package header are not visible outside the package body.
The names of procedures and functions defined in the package body must be unique among the names of procedures and functions defined in the package header and implemented in the package body.

**Package procedure and function names may shadow global routines**

If a package header or package body declares a procedure or function with the same name as a stored procedure or function in the global namespace, it is not possible to call that global procedure or function from the package body. In this case, the procedure or function of the package will always be called.

For this reason, it is recommended that the names of stored procedures and functions in packages do not overlap with names of stored procedures and functions in the global namespace.

**Rules**

- In the package body, all procedures and functions must be implemented with the same signature as declared in the header and at the beginning of the package body.
- The default values for procedure or function parameters cannot be overridden (as specified in the package header or in `<package_item>`). This means default values can only be defined in `<package_body_item>` for procedures or functions that have not been defined in the package header or earlier in the package body.

**Who Can Create a Package Body**

The `CREATE PACKAGE BODY` statement can be executed by:

- **Administrators**
- The owner of the package
- Users with the `ALTER ANY PACKAGE` privilege

**Examples of CREATE PACKAGE BODY**

*Creating the package body*

```sql
CREATE PACKAGE BODY APP_VAR
AS
BEGIN
  -- Returns the start date of the period
  FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC
  AS
  BEGIN
    RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEBEGIN');
  END
  -- Returns the end date of the period
  FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC
```
AS
BEGIN
   RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEEND');
END
-- Sets the date range of the working period
PROCEDURE SET_DATERANGE(ADATEBEGIN DATE, ADATEEND DATE)
AS
BEGIN
   RDB$SET_CONTEXT('USER_SESSION', 'DATEBEGIN', ADATEBEGIN);
   RDB$SET_CONTEXT('USER_SESSION', 'DATEEND', ADATEEND);
END
END

See also
DROP PACKAGE BODY, RECREATE PACKAGE BODY, CREATE PACKAGE

5.12.2. DROP PACKAGE BODY

Used for
Dropping a package body

Available in
DSQL

Syntax

DROP PACKAGE package_name

Table 52. DROP PACKAGE BODY Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>package_name</td>
<td>Package name</td>
</tr>
</tbody>
</table>

The DROP PACKAGE BODY statement deletes the package body.

Who Can Drop a Package Body

The DROP PACKAGE BODY statement can be executed by:

- Administrators
- The owner of the package
- Users with the ALTER ANY PACKAGE privilege

Examples of DROP PACKAGE BODY

Dropping the package body

DROP PACKAGE BODY APP_VAR;
RECREATE PACKAGE BODY

Used for
Creating a new or recreating an existing package body

Available in
DSQL

Syntax

RECREATE PACKAGE BODY name
AS
BEGIN
[ <package_item> ... ]
[ <package_body_item> ... ]
END

!! See syntax of CREATE PACKAGE BODY for further rules !!

The RECREATE PACKAGE BODY statement creates a new or recreates an existing package body. If a package body with the same name already exists, the statement will try to drop it and then create a new package body. After recreating the package body, privileges of the package and its routines are preserved.

See CREATE PACKAGE BODY for more details.

Examples of RECREATE PACKAGE BODY

Recreating the package body

RECREATE PACKAGE BODY APP_VAR
AS
BEGIN
-- Returns the start date of the period
FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC
AS
BEGIN
RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEBEGIN');
END
-- Returns the end date of the period
FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC
AS
BEGIN
RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEEND');
END
-- Sets the date range of the working period
PROCEDURE SET_DATERANGE(ADATEBEGIN DATE, ADATEEND DATE)
A BLOB FILTER is a database object that is a special type of external function, with the sole purpose of taking a BLOB object in one format and converting it to a BLOB object in another format. The formats of the BLOB objects are specified with user-defined BLOB subtypes.

External functions for converting BLOB types are stored in dynamic libraries and loaded when necessary.

For more details on BLOB subtypes, see Binary Data Types.

### 5.13.1. DECLARE FILTER

#### Used for
Declaring a BLOB filter to the database

#### Available in
DSQL, ESQL

#### Syntax

```
DECLARE FILTER filtername
    INPUT_TYPE <sub_type> OUTPUT_TYPE <sub_type>
    ENTRY_POINT 'function_name' MODULE_NAME 'library_name'
```

#### Parameter Descriptions

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filtername</td>
<td>Filter name in the database. It may consist of up to 31 characters. It need not be the same name as the name exported from the filter library via ENTRY_POINT.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>sub_type</td>
<td>BLOB subtype</td>
</tr>
<tr>
<td>number</td>
<td>BLOB subtype number (must be negative)</td>
</tr>
<tr>
<td>mnemonic</td>
<td>BLOB subtype mnemonic name</td>
</tr>
<tr>
<td>function_name</td>
<td>The exported name (entry point) of the function</td>
</tr>
<tr>
<td>library_name</td>
<td>The name of the module where the filter is located</td>
</tr>
<tr>
<td>user_defined</td>
<td>User-defined BLOB subtype mnemonic name</td>
</tr>
</tbody>
</table>

The DECLARE FILTER statement makes a BLOB filter available to the database. The name of the BLOB filter must be unique among the names of BLOB filters.

**Specifying the Subtypes**

The subtypes can be specified as the subtype number or as the subtype mnemonic name. Custom subtypes must be represented by negative numbers (from -1 to -32,768). An attempt to declare more than one BLOB filter with the same combination of the input and output types will fail with an error.

**INPUT_TYPE**

clause defining the BLOB subtype of the object to be converted

**OUTPUT_TYPE**

clause defining the BLOB subtype of the object to be created.

Mnemonic names can be defined for custom BLOB subtypes and inserted manually into the system table RDB$TYPES system table:

```
INSERT INTO RDB$TYPES (RDB$FIELD_NAME, RDB$TYPE, RDB$TYPE_NAME)
VALUES ('RDB$FIELD_SUB_TYPE', -33, 'MIDI');
```

After the transaction is committed, the mnemonic names can be used in declarations when you create new filters.

The value of the column RDB$FIELD_NAME must always be 'RDB$FIELD_SUB_TYPE'. If a mnemonic names was defined in upper case, they can be used case-insensitively and without quotation marks when a filter is declared, following the rules for other object names.

**Warning**

From Firebird 3.0 onward, the system tables will no longer be writable by users. However, inserting custom types into RDB$TYPES is still possible. Firebird 4 will introduce a system privilege CREATE_USER_TYPES for creating custom subtypes.

**Parameters**
**ENTRY_POINT**

clause defining the name of the entry point (the name of the imported function) in the module.

**MODULE_NAME**

The clause defining the name of the module where the exported function is located. By default, modules must be located in the UDF folder of the root directory on the server. The UDFAccess parameter in firebird.conf allows editing of access restrictions to filter libraries.

Any user connected to the database can declare a BLOB filter.

**Who Can Create a BLOB Filter?**

The DECLARE FILTER statement can be executed by:

- **Administrators**
- **Users with the CREATE FILTER privilege**

The user executing the DECLARE FILTER statement becomes the owner of the filter.

**Examples of DECLARE FILTER**

1. Creating a BLOB filter using subtype numbers.

   ```plaintext
   DECLARE FILTER DESC_FILTER
   INPUT_TYPE 1
   OUTPUT_TYPE -4
   ENTRY_POINT 'desc_filter'
   MODULE_NAME 'FILTERLIB';
   ```

2. Creating a BLOB filter using subtype mnemonic names.

   ```plaintext
   DECLARE FILTER FUNNEL
   INPUT_TYPE blr OUTPUT_TYPE text
   ENTRY_POINT 'blr2asc' MODULE_NAME 'myfilterlib';
   ```

**See also**

DROP FILTER

**5.13.2. DROP FILTER**

*Used for*

Removing a BLOB filter declaration from the database

*Available in*

DSQL, ESQL
Syntax

```
DROP FILTER filtername
```

### Table 54. DROP FILTER Statement Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filtername</td>
<td>Filter name in the database</td>
</tr>
</tbody>
</table>

The `DROP FILTER` statement removes the declaration of a BLOB filter from the database. Removing a BLOB filter from a database makes it unavailable for use from that database. The dynamic library where the conversion function is located remains intact and the removal from one database does not affect other databases in which the same BLOB filter is still declared.

**Who Can Drop a BLOB Filter?**

The `DROP FILTER` statement can be executed by:

- Administrators
- The owner of the filter
- Users with the `DROP ANY FILTER` privilege

**DROP FILTER Example**

*Dropping a BLOB filter.*

```
DROP FILTER DESC_FILTER;
```

**See also**

`DECLARE FILTER`

### 5.14. SEQUENCE (GENERATOR)

A sequence or a generator is a database object used to get unique number values to fill a series. “Sequence” is the SQL-compliant term for the same thing which, in Firebird, has traditionally been known as “generator”. Firebird has syntax for both terms.

Sequences (or generators) are always stored as 64-bit integers, regardless of the SQL dialect of the database.

If a client is connected using Dialect 1, the server sends sequence values to it as 32-bit integers. Passing a sequence value to a 32-bit field or variable will not cause errors as long as the current value of the sequence does not exceed the limits of a 32-bit number. However, as soon as the sequence value exceeds this limit, a database in Dialect 3 will produce an error. A database in Dialect 1 will keep truncating the values, which will compromise the uniqueness of the series.
This section describes how to create, alter, set and drop sequences.

### 5.14.1. CREATE SEQUENCE

**Used for**

Creating a new SEQUENCE (GENERATOR)

**Available in**

DSQL, ESQL

**Syntax**

```
CREATE {SEQUENCE | GENERATOR} seq_name
[START WITH start_value]
[INCREMENT [BY] increment]
```

**Table 55. CREATE SEQUENCE Statement Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq_name</td>
<td>Sequence (generator) name. It may consist of up to 31 characters</td>
</tr>
<tr>
<td>start_value</td>
<td>Initial value of the sequence</td>
</tr>
<tr>
<td>increment</td>
<td>Increment of the sequence (when using NEXT VALUE FOR seq_name); cannot be 0</td>
</tr>
</tbody>
</table>

The statements `CREATE SEQUENCE` and `CREATE GENERATOR` are synonymous—both create a new sequence. Either can be used, but `CREATE SEQUENCE` is recommended as that is the syntax defined in the SQL standard.

When a sequence is created, its value is set to the value specified in the option `START WITH` clause. If there is no `START WITH` clause, then the sequence is set to 0.

The optional `INCREMENT [BY]` clause allows you to specify an increment for the `NEXT VALUE FOR seq_name` expression. By default, the increment is 1 (one). The increment cannot be set to 0 (zero). The `GEN_ID(seq_name, <step>)` function can be called instead, to “step” the series by a different integer number. The increment specified through `INCREMENT [BY]` is not used for `GEN_ID`.

---

**Bug with START WITH and INCREMENT [BY]**

The SQL standard specifies that the `START WITH` clause should specify the initial value generated on the first call to `NEXT VALUE FOR seq_name`, but instead Firebird uses it to set the current value of the sequence. As a result the first call to `NEXT VALUE FOR seq_name` incorrectly generates the value `start_value + increment`.

Creating a sequence without a `START WITH` clause is currently equivalent to specifying `START WITH 0`, while it should be equivalent to `START WITH 1`.

This will be fixed in Firebird 4, see also CORE-6084

---

**Non-standard behaviour for negative increments**

---

208
The SQL standard specifies that sequences with a negative increment should start at the maximum value of the sequence ($2^{63} - 1$) and count down. Firebird does not do that, and instead starts at $0 + \text{increment}$.

This may change in a future Firebird version.

**Who Can Create a Sequence?**

The `CREATE SEQUENCE` (CREATE GENERATOR) statement can be executed by:

- Administrators
- Users with the `CREATE SEQUENCE` (CREATE GENERATOR) privilege

The user executing the `CREATE SEQUENCE` (CREATE GENERATOR) statement becomes its owner.

**Examples of CREATE SEQUENCE**

1. Creating the `EMP_NO_GEN` sequence using `CREATE SEQUENCE`.

   ```sql
   CREATE SEQUENCE EMP_NO_GEN;
   ```

2. Creating the `EMP_NO_GEN` sequence using `CREATE GENERATOR`.

   ```sql
   CREATE GENERATOR EMP_NO_GEN;
   ```

3. Creating the `EMP_NO_GEN` sequence with an initial value of 5 and an increment of 1. See note Bug with `START WITH` and `INCREMENT [BY]`.

   ```sql
   CREATE SEQUENCE EMP_NO_GEN START WITH 5;
   ```

4. Creating the `EMP_NO_GEN` sequence with an initial value of 1 and an increment of 10. See note Bug with `START WITH` and `INCREMENT [BY]`.

   ```sql
   CREATE SEQUENCE EMP_NO_GEN INCREMENT BY 10;
   ```

5. Creating the `EMP_NO_GEN` sequence with an initial value of 5 and an increment of 10. See note Bug with `START WITH` and `INCREMENT [BY]`.

   ```sql
   CREATE SEQUENCE EMP_NO_GEN START WITH 5 INCREMENT BY 10;
   ```

See also

- ALTER SEQUENCE, CREATE OR ALTER SEQUENCE, DROP SEQUENCE, RECREATE SEQUENCE, SET GENERATOR, NEXT VALUE FOR, GEN_ID() function
5.14.2. ALTER SEQUENCE

**Used for**
Setting the value of a sequence or generator to a specified value

**Available in**
DSQL

**Syntax**

```
ALTER {SEQUENCE | GENERATOR} seq_name
  [RESTART [WITH newvalue]]
  [INCREMENT [BY] increment]
```

**Table 56. ALTER SEQUENCE Statement Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq_name</td>
<td>Sequence (generator) name</td>
</tr>
<tr>
<td>newvalue</td>
<td>New sequence (generator) value. A 64-bit integer from (-2^{63}) to (2^{63}-1).</td>
</tr>
<tr>
<td>increment</td>
<td>Increment of the sequence (when using NEXT VALUE FOR seq_name); cannot be 0</td>
</tr>
</tbody>
</table>

The ALTER SEQUENCE statement sets the current value of a sequence or generator to the specified value and/or changes the increment of the sequence.

The RESTART WITH newvalue clause allows you to set the value of a sequence. The RESTART clause (without WITH) restarts the sequence with the initial value configured using the START WITH clause when the sequence was created.

**Bugs with** RESTART

The initial value (either saved in the metadata or specified in the WITH clause) is used to set the current value of the sequence, instead of the next value generated as required by the SQL standard. See note Bug with START WITH and INCREMENT [BY] for more information.

In addition, RESTART WITH newvalue will not only restart the sequence with the specified value, but also store newvalue as the new initial value of the sequence. This means that a subsequent ALTER SEQUENCE RESTART will also use newvalue. This behaviour does not match the behaviour specified in the SQL standard.

This bug will be fixed in Firebird 4, see also CORE-6386

**Warning**

Incorrect use of the ALTER SEQUENCE statement (changing the current value of the sequence or generator) is likely to break the logical integrity of data.

INCREMENT [BY] allows you to change the sequence increment for the NEXT VALUE FOR expression.
Changing the increment value takes effect for all queries that run after the transaction commits. Procedures that are called for the first time after changing the commit, will use the new value if they use NEXT VALUE FOR. Procedures that were already used (and cached in the metadata cache) will continue to use the old increment. You may need to close all connections to the database for the metadata cache to clear, and the new increment to be used. Procedures using NEXT VALUE FOR do not need to be recompiled to see the new increment. Procedures using GEN_ID(gen, expression) are not affected when the increment is changed.

Who Can Alter a Sequence?

The ALTER SEQUENCE (ALTER GENERATOR) statement can be executed by:

- Administrators
- The owner of the sequence
- Users with the ALTER ANY SEQUENCE (ALTER ANY GENERATOR) privilege

Examples of ALTER SEQUENCE

1. Setting the value of the EMP_NO_GEN sequence to 145.

   ALTER SEQUENCE EMP_NO_GEN RESTART WITH 145;

2. Resetting the base value of the sequence EMP_NO_GEN to the initial value stored in the metadata

   ALTER SEQUENCE EMP_NO_GEN RESTART;

3. Changing the increment of sequence EMP_NO_GEN to 10

   ALTER SEQUENCE EMP_NO_GEN INCREMENT BY 10;

See also

SET GENERATOR, CREATE SEQUENCE, CREATE OR ALTER SEQUENCE, DROP SEQUENCE, RECREATE SEQUENCE, NEXT VALUE FOR, GEN_ID() function

5.14.3. CREATE OR ALTER SEQUENCE

Used for
Creating a new or modifying an existing sequence

Available in
DSQL, ESQL
Syntax

CREATE OR ALTER {SEQUENCE | GENERATOR} seq_name
{RESTART | START WITH start_value}
[INCREMENT [BY] increment]

Table 57. CREATE OR ALTER SEQUENCE Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq_name</td>
<td>Sequence (generator) name. It may consist of up to 31 characters</td>
</tr>
<tr>
<td>start_value</td>
<td>Initial value of the sequence</td>
</tr>
<tr>
<td>increment</td>
<td>Increment of the sequence (when using NEXT VALUE FOR seq_name); cannot be 0</td>
</tr>
</tbody>
</table>

If the sequence does not exist, it will be created. An existing sequence will be changed:

- If RESTART is specified, the sequence will restarted with the initial value stored in the metadata
- If the START WITH clause is specified, start_value is stored as the initial value in the metadata, and the sequence is restarted
- If the INCREMENT [BY] clause is specified, increment is stored as the increment in the metadata, and used for subsequent calls to NEXT VALUE FOR

Example of CREATE OR ALTER SEQUENCE

Create a new or modify an existing sequence EMP_NO_GEN

```
CREATE OR ALTER SEQUENCE EMP_NO_GEN
    START WITH 10
    INCREMENT BY 1
```

See also

CREATE SEQUENCE, ALTER SEQUENCE, DROP SEQUENCE, RECREATE SEQUENCE, SET GENERATOR, NEXT VALUE FOR, GEN_ID() function

5.14.4. DROP SEQUENCE

Used for
Dropping (deleting) a SEQUENCE (GENERATOR)

Available in
DSQL, ESQL

Syntax

DROP {SEQUENCE | GENERATOR} seq_name

Table 58. DROP SEQUENCE Statement Parameter
The statements DROP SEQUENCE and DROP GENERATOR statements are equivalent: both drop (delete) an existing sequence (generator). Either is valid but DROP SEQUENCE, being defined in the SQL standard, is recommended.

The statements will fail if the sequence (generator) has dependencies.

Who Can Drop a Sequence?

The DROP SEQUENCE (DROP GENERATOR) statement can be executed by:

- Administrators
- The owner of the sequence
- Users with the DROP ANY SEQUENCE (DROP ANY GENERATOR) privilege

Example of DROP SEQUENCE

Dropping the EMP_NO_GEN series:

```sql
DROP SEQUENCE EMP_NO_GEN;
```

See also

CREATE SEQUENCE, CREATE OR ALTER SEQUENCE, RECREATE SEQUENCE

5.14.5. RECREATE SEQUENCE

Used for
Creating or recreating a sequence (generator)

Available in
DSQL, ESQL

Syntax

```
RECREATE {SEQUENCE | GENERATOR} seq_name
[START WITH start_value]
[INCREMENT [BY] increment]
```

Table 59. RECREATE SEQUENCE Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq_name</td>
<td>Sequence (generator) name. It may consist of up to 31 characters</td>
</tr>
<tr>
<td>start_value</td>
<td>Initial value of the sequence</td>
</tr>
</tbody>
</table>
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>increment</td>
<td>Increment of the sequence (when using \texttt{NEXT VALUE FOR seq_name}); cannot be 0</td>
</tr>
</tbody>
</table>

See \texttt{CREATE SEQUENCE} for the full syntax of \texttt{CREATE SEQUENCE} and descriptions of defining a sequence and its options.

\texttt{RECREATE SEQUENCE} creates or recreates a sequence. If a sequence with this name already exists, the \texttt{RECREATE SEQUENCE} statement will try to drop it and create a new one. Existing dependencies will prevent the statement from executing.

**Example of \texttt{RECREATE SEQUENCE}**

**Recreating sequence \texttt{EMP\_NO\_GEN}**

```
RECREATE SEQUENCE EMP\_NO\_GEN
  START WITH 10
  INCREMENT BY 2;
```

*See also*

\texttt{CREATE SEQUENCE, ALTER SEQUENCE, CREATE OR ALTER SEQUENCE, DROP SEQUENCE, SET GENERATOR, NEXT VALUE FOR, GEN\_ID()} function

### 5.14.6. \texttt{SET GENERATOR}

**Used for**

Setting the value of a sequence or generator to a specified value

**Available in**

DSQL, ESQL

**Syntax**

```
SET GENERATOR seq\_name TO new\_val
```

**Table 60. \texttt{SET GENERATOR} Statement Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq_name</td>
<td>Generator (sequence) name</td>
</tr>
<tr>
<td>new_val</td>
<td>New sequence (generator) value. A 64-bit integer from $-2^{63}$ to $2^{63}$-1.</td>
</tr>
</tbody>
</table>

The \texttt{SET GENERATOR} statement sets the current value of a sequence or generator to the specified value.

Although \texttt{SET GENERATOR} is considered outdated, it is retained for backward compatibility. Use of the standards-compliant \texttt{ALTER SEQUENCE} is recommended.
Who Can Use a SET GENERATOR?

The SET GENERATOR statement can be executed by:

- Administrators
- The owner of the sequence (generator)
- Users with the ALTER ANY SEQUENCE (ALTER ANY GENERATOR) privilege

Example of SET GENERATOR

Setting the value of the EMP_NO_GEN sequence to 145:

```
SET GENERATOR EMP_NO_GEN TO 145;
```

The same can be done with ALTER SEQUENCE:

```
ALTER SEQUENCE EMP_NO_GEN RESTART WITH 145;
```

See also

ALTER SEQUENCE, CREATE SEQUENCE, CREATE OR ALTER SEQUENCE, DROP SEQUENCE, NEXT VALUE FOR, GEN_ID() function

5.15. EXCEPTION

This section describes how to create, modify and delete custom exceptions for use in error handlers in PSQL modules.

5.15.1. CREATE EXCEPTION

Used for
Creating a new exception for use in PSQL modules

Available in
DSQL, ESQL

Syntax

```
CREATE EXCEPTION exception_name '<message>'

<message> ::= <message-part> [<message-part> ...]

<message-part> ::=<text>
                 | @<slot>

<slot> ::= one of 1..9
```
Table 61. CREATE EXCEPTION Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception_name</td>
<td>Exception name. The maximum length is 31 characters</td>
</tr>
<tr>
<td>message</td>
<td>Default error message. The maximum length is 1,021 characters</td>
</tr>
<tr>
<td>text</td>
<td>Text of any character</td>
</tr>
<tr>
<td>slot</td>
<td>Slot number of a parameter. Numbering starts at 1. Maximum slot number is 9.</td>
</tr>
</tbody>
</table>

The statement CREATE EXCEPTION creates a new exception for use in PSQL modules. If an exception with the same name exists, the statement will fail with an appropriate error message.

The exception name is a standard identifier. In a Dialect 3 database, it can be enclosed in double quotes to make it case-sensitive and, if required, to use characters that are not valid in regular identifiers. See Identifiers for more information.

The default message is stored in character set NONE, i.e. in characters of any single-byte character set. The text can be overridden in the PSQL code when the exception is thrown.

The error message may contain “parameter slots” that can be filled when raising the exception.

- If the message contains a parameter slot number that is greater than 9, the second and subsequent digits will be treated as literal text. For example @10 will be interpreted as slot 1 followed by a literal ‘0’.

- Custom exceptions are stored in the system table RDB$EXCEPTIONS.

Who Can Create an Exception

The CREATE EXCEPTION statement can be executed by:

- Administrators
- Users with the CREATE EXCEPTION privilege

The user executing the CREATE EXCEPTION statement becomes the owner of the exception.

CREATE EXCEPTION Examples

Creating an exception named E_LARGE_VALUE

```sql
CREATE EXCEPTION E_LARGE_VALUE
    'The value is out of range';
```

Creating a parameterized exception E_INVALID_VALUE

```sql
CREATE EXCEPTION E_INVALID_VALUE
    'Invalid value @1 for field @2';
```
Tips

Grouping CREATE EXCEPTION statements together in system update scripts will simplify working with them and documenting them. A system of prefixes for naming and categorising groups of exceptions is recommended.

See also

ALTER EXCEPTION, CREATE OR ALTER EXCEPTION, DROP EXCEPTION, RECREATE EXCEPTION

5.15.2. ALTER EXCEPTION

Used for

Modifying the message returned from a custom exception

Available in

DSQL, ESQL

Syntax

```
ALTER EXCEPTION exception_name '<message>'
```

!! See syntax of CREATE EXCEPTION for further rules !!

The statement ALTER EXCEPTION can be used at any time, to modify the default text of the message.

Who Can Alter an Exception

The ALTER EXCEPTION statement can be executed by:

- Administrators
- The owner of the exception
- Users with the ALTER ANY EXCEPTION privilege

ALTER EXCEPTION Examples

Changing the default message for the exception E_LARGE_VALUE

```
ALTER EXCEPTION E_LARGE_VALUE
' The value exceeds the prescribed limit of 32,765 bytes';
```

See also

CREATE EXCEPTION, CREATE OR ALTER EXCEPTION, DROP EXCEPTION, RECREATE EXCEPTION

5.15.3. CREATE OR ALTER EXCEPTION

Used for

Modifying the message returned from a custom exception, if the exception exists; otherwise, creating a new exception
Available in
DSQL

Syntax

```
CREATE OR ALTER EXCEPTION exception_name '<message>'
```

!! See syntax of CREATE EXCEPTION for further rules !!

The statement CREATE OR ALTER EXCEPTION is used to create the specified exception if it does not exist, or to modify the text of the error message returned from it if it exists already. If an existing exception is altered by this statement, any existing dependencies will remain intact.

**CREATE OR ALTER EXCEPTION Example**

*Changing the message for the exception E_LARGE_VALUE*

```
CREATE OR ALTER EXCEPTION E_LARGE_VALUE
   'The value is higher than the permitted range 0 to 32,765';
```

See also

CREATE EXCEPTION, ALTER EXCEPTION, RECREATE EXCEPTION

**5.15.4. DROP EXCEPTION**

*Used for*
Deleting a custom exception

*Available in*
DSQL, ESQL

**Syntax**

```
DROP EXCEPTION exception_name
```

*Table 62. DROP EXCEPTION Statement Parameter*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception_name</td>
<td>Exception name</td>
</tr>
</tbody>
</table>

The statement DROP EXCEPTION is used to delete an exception. Any dependencies on the exception will cause the statement to fail, and the exception will not be deleted.

**Who Can Drop an Exception**

The DROP EXCEPTION statement can be executed by:

- Administrators
• The owner of the exception
• Users with the DROP ANY EXCEPTION privilege

**DROP EXCEPTION Examples**

*dropping exception E_LARGE_VALUE*

```sql
DROP EXCEPTION E_LARGE_VALUE;
```

**See also**

CREATE EXCEPTION, RECREATE EXCEPTION

**5.15.5. RECREATE EXCEPTION**

*used for*

Creating a new custom exception or recreating an existing one

*available in*

DSQL

**Syntax**

```sql
RECREATE EXCEPTION exception_name '<message>'
```

!! See syntax of CREATE EXCEPTION for further rules !!

The statement RECREATE EXCEPTION creates a new exception for use in PSQL modules. If an exception with the same name exists already, the RECREATE EXCEPTION statement will try to drop it and create a new one. If there are any dependencies on the existing exception, the attempted deletion fails and RECREATE EXCEPTION is not executed.

**RECREATE EXCEPTION Example**

*recreating the E_LARGE_VALUE exception*

```sql
RECREATE EXCEPTION E_LARGE_VALUE
    'The value exceeds its limit';
```

**See also**

CREATE EXCEPTION, DROP EXCEPTION, CREATE OR ALTER EXCEPTION

**5.16. COLLATION**

In SQL, text strings are sortable objects. This means that they obey ordering rules, such as alphabetical order. Comparison operations can be applied to such text strings (for example, “less than” or “greater than”), where the comparison must apply a certain sort order or collation. For example, the expression “'a' < 'b'” means that “'a'” precedes “'b'” in the collation. The expression
“'c' > 'b'” means that ‘c’ follows ‘b’ in the collation. Text strings of more than one character are sorted using sequential character comparisons: first the first characters of the two strings are compared, then the second characters, and so on, until a difference is found between the two strings. This difference defines the sort order.

A COLLATION is the schema object that defines a collation (or sort order).

**5.16.1. CREATE COLLATION**

*Used for*

Creating a new collation for a supported character set available to the database

*Available in*

DSQL

*Syntax*

```sql
CREATE COLLATION collname
    FOR charset
     [FROM {basecoll | EXTERNAL ('extname')}] [NO PAD | PAD SPACE] [CASE [IN]SENSITIVE] [ACCENT [IN]SENSITIVE] ['<specific-attributes>' ]

<specific-attributes> ::= <attribute> [ ; <attribute> ...]

<attribute> ::= attrname=attrvalue
```

*Table 63. CREATE COLLATION Statement Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collname</td>
<td>The name to use for the new collation. The maximum length is 31 characters</td>
</tr>
<tr>
<td>charset</td>
<td>A character set present in the database</td>
</tr>
<tr>
<td>basecoll</td>
<td>A collation already present in the database</td>
</tr>
<tr>
<td>extname</td>
<td>The collation name used in the .conf file</td>
</tr>
</tbody>
</table>

The CREATE COLLATION statement does not “create” anything, its purpose is to make a collation known to a database. The collation must already be present on the system, typically in a library file, and must be properly registered in a .conf file in the int\l subdirectory of the Firebird installation.

The collation may alternatively be based on one that is already present in the database.

**How the Engine Detects the Collation**

The optional FROM clause specifies the base collation that is used to derive a new collation. This collation must already be present in the database. If the keyword EXTERNAL is specified, then
Firebird will scan the `.conf` files in `$fbroot/intl/`, where `extname` must exactly match the name in the configuration file (case-sensitive).

If no `FROM` clause is present, Firebird will scan the `.conf` file(s) in the `intl` subdirectory for a collation with the collation name specified in `CREATE COLLATION`. In other words, omitting the `FROM basecoll` clause is equivalent to specifying `FROM EXTERNAL ('collname')`.

The — single-quoted — `extname` is case-sensitive and must correspond exactly with the collation name in the `.conf` file. The `collname`, `charset` and `basecoll` parameters are case-insensitive unless enclosed in double-quotes.

When creating a collation, you can specify whether trailing spaces are included in the comparison. If the `NO PAD` clause is specified, trailing spaces are taken into account in the comparison. If the `PAD SPACE` clause is specified, trailing spaces are ignored in the comparison.

The optional `CASE` clause allows you to specify whether the comparison is case-sensitive or case-insensitive.

The optional `ACCENT` clause allows you to specify whether the comparison is accent-sensitive or accent-insensitive (e.g. if ‘e’ and ‘é’ are considered equal or unequal).

### Specific Attributes

The `CREATE COLLATION` statement can also include specific attributes to configure the collation. The available specific attributes are listed in the table below. Not all specific attributes apply to every collation. If the attribute is not applicable to the collation, but is specified when creating it, it will not cause an error.

⚠️ Specific attribute names are case sensitive.

In the table, “1 bpc” indicates that an attribute is valid for collations of character sets using 1 byte per character (so-called narrow character sets), and “UNI” for “Unicode collations”.

#### Table 64. Specific Collation Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Values</th>
<th>Valid for</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISABLE-COMPRESSIONS</td>
<td>0, 1</td>
<td>1 bpc</td>
<td>Disables compressions (a.k.a. contractions). Compressions cause certain character sequences to be sorted as atomic units, e.g. Spanish c+h as a single character ch.</td>
</tr>
<tr>
<td>DISABLE-EXPANSIONS</td>
<td>0, 1</td>
<td>1 bpc</td>
<td>Disables expansions. Expansions cause certain characters (e.g. ligatures or umlauted vowels) to be treated as character sequences and sorted accordingly.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Values</td>
<td>Valid for</td>
<td>Comment</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>-----------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ICU-VERSION</td>
<td>default or</td>
<td>UNI</td>
<td>Specifies the ICU library version to use. Valid values are the ones defined in the applicable <code>&lt;intl_module&gt;</code> element in <code>intl/fbintl.conf</code>. Format: either the string literal “default” or a major+minor version number like “3.0” (both unquoted).</td>
</tr>
<tr>
<td>LOCALE</td>
<td>xx_YY</td>
<td>UNI</td>
<td>Specifies the collation locale. Requires complete version of ICU libraries. Format: a locale string like “du_NL” (unquoted)</td>
</tr>
<tr>
<td>MULTI-LEVEL</td>
<td>0, 1</td>
<td>1 bpc</td>
<td>Uses more than one ordering level</td>
</tr>
<tr>
<td>NUMERIC-SORT</td>
<td>0, 1</td>
<td>UNI</td>
<td>Treats contiguous groups of decimal digits in the string as atomic units and sorts them numerically. (This is also known as natural sorting)</td>
</tr>
<tr>
<td>SPECIALS-FIRST</td>
<td>0, 1</td>
<td>1 bpc</td>
<td>Orders special characters (spaces, symbols etc.) before alphanumeric characters</td>
</tr>
</tbody>
</table>

If you want to add a new character set with its default collation into your database, declare and run the stored procedure `sp_register_character_set(name, max_bytes_per_character)`, found in `misc/intl.sql` under the Firebird installation directory.

In order for this to work, the character set must be present on the system and registered in a `.conf` file in the `intl` subdirectory.

Who Can Create a Collation

The CREATE COLLATION statement can be executed by:

- **Administrators**
- **Users with the CREATE COLLATION privilege**

The user executing the CREATE COLLATION statement becomes the owner of the collation.

Examples using CREATE COLLATION

1. Creating a collation using the name found in the `fbintl.conf` file (case-sensitive)

   ```sql
   CREATE COLLATION ISO8859_1_UNICODE FOR ISO8859_1;
   ```

2. Creating a collation using a special (user-defined) name (the “external” name must completely
match the name in the fbintl.conf file)

```
CREATE COLLATION LAT_UNI
 FOR ISO8859_1
 FROM EXTERNAL ("ISO8859_1_UNICODE");
```

3. Creating a case-insensitive collation based on one already existing in the database

```
CREATE COLLATION ES_ES_NOPAD_CI
 FOR ISO8859_1
 FROM ES_ES
 NO PAD
 CASE INSENSITIVE;
```

4. Creating a case-insensitive collation based on one already existing in the database with specific attributes

```
CREATE COLLATION ES_ES_CI_COMPR
 FOR ISO8859_1
 FROM ES_ES
 CASE INSENSITIVE
 'DISABLE-COMPRESSIONS=0';
```

5. Creating a case-insensitive collation by the value of numbers (the so-called natural collation)

```
CREATE COLLATION nums_coll FOR UTF8
 FROM UNICODE
 CASE INSENSITIVE 'NUMERIC-SORT=1';

CREATE DOMAIN dm_nums AS varchar(20)
 CHARACTER SET UTF8 COLLATE nums_coll; -- original (manufacturer) numbers

CREATE TABLE wares(id int primary key, articul dm_nums ...);
```

See also

DROP COLLATION

### 5.16.2. DROP COLLATION

**Used for**

Removing a collation from the database

**Available in**

DSQL
Syntax

```
DROP COLLATION collname
```

**Table 65. DROP COLLATION Statement Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collname</td>
<td>The name of the collation</td>
</tr>
</tbody>
</table>

The DROP COLLATION statement removes the specified collation from the database, if it exists. An error will be raised if the specified collation is not present.

If you want to remove an entire character set with all its collations from the database, declare and execute the stored procedure `sp_unregister_character_set(name)` from the `misc/intl.sql` subdirectory of the Firebird installation.

**Who Can Drop a Collation**

The Drop COLLATION statement can be executed by:

- **Administrators**
- The owner of the collation
- Users with the DROP ANY COLLATION privilege

**Example using DROP COLLATION**

*Deleting the ES_ES_NOPAD_CI collation.*

```
DROP COLLATION ES_ES_NOPAD_CI;
```

**See also**

CREATE COLLATION

---

**5.17. CHARACTER SET**

**5.17.1. ALTER CHARACTER SET**

*Used for*

Setting the default collation for a character set

*Available in*

DSQL

**Syntax**

```
ALTER CHARACTER SET charset
```
Table 66. ALTER CHARACTER SET Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charset</td>
<td>Character set identifier</td>
</tr>
<tr>
<td>collation</td>
<td>The name of the collation</td>
</tr>
</tbody>
</table>

The ALTER CHARACTER SET statement changes the default collation for the specified character set. It will affect the future usage of the character set, except for cases where the COLLATE clause is explicitly overridden. In that case, the collation sequence of existing domains, columns and PSQL variables will remain intact after the change to the default collation of the underlying character set.

If you change the default collation for the database character set (the one defined when the database was created), it will change the default collation for the database.

If you change the default collation for the character set that was specified during the connection, string constants will be interpreted according to the new collation value, except in those cases where the character set and/or the collation have been overridden.

Who Can Alter a Character Set

The ALTER CHARACTER SET statement can be executed by:

- Administrators
- Users with the ALTER ANY CHARACTER SET privilege

ALTER CHARACTER SET Example

Setting the default UNICODE_CI_AI collation for the UTF8 encoding

```
ALTER CHARACTER SET UTF8
SET DEFAULT COLLATION UNICODE_CI_AI;
```

5.18. Comments

Database objects and a database itself may be annotated with comments. It is a convenient mechanism for documenting the development and maintenance of a database. Comments created with COMMENT ON will survive a gbak backup and restore.

5.18.1. COMMENT ON

Used for
Documenting metadata
Available in

DSQL

Syntax

```
COMMENT ON <object> IS {"sometext" | NULL}

<object> ::= {DATABASE | SCHEMA} | <basic-type> objectname | COLUMN relationname.fieldname | [{PROCEDURE | FUNCTION}] PARAMETER [packagename.]routinename.paramname | {PROCEDURE | [EXTERNAL] FUNCTION} [package_name.]routinename

<basic-type> ::= CHARACTER SET | COLLATION | DOMAIN | EXCEPTION | FILTER | GENERATOR | INDEX | PACKAGE | ROLE | SEQUENCE | TABLE | TRIGGER | USER | VIEW
```

Table 67. COMMENT ON Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sometext</td>
<td>Comment text</td>
</tr>
<tr>
<td>basic-type</td>
<td>Metadata object type</td>
</tr>
<tr>
<td>objectname</td>
<td>Metadata object name</td>
</tr>
<tr>
<td>relationname</td>
<td>Name of table or view</td>
</tr>
<tr>
<td>fieldname</td>
<td>Name of the column</td>
</tr>
<tr>
<td>package_name</td>
<td>Name of the package</td>
</tr>
<tr>
<td>routinename</td>
<td>Name of stored procedure or function</td>
</tr>
<tr>
<td>paramname</td>
<td>Name of a stored procedure or function parameter</td>
</tr>
</tbody>
</table>

The `COMMENT ON` statement adds comments for database objects (metadata). Comments are saved to the `RDB$DESCRIPTION` column of the corresponding system tables. Client applications can view comments from these fields.

1. If you add an empty comment (""""), it will be saved as NULL in the database.
2. The `COMMENT ON USER` statement will only create comments on users managed by the default usermanager (the first plugin listed in the `UserManager` config option). See also [CORE-6479](#).
3. Comments on users are stored in the security database.
4. `SCHEMA` is currently a synonym for `DATABASE`; this may change in a future
Who Can Add a Comment

The COMMENT ON statement can be executed by:

- Administrators
- The owner of the object that is commented on
- Users with the ALTER ANY object_type privilege, where object_type is the type of object commented on (e.g. PROCEDURE)

Examples using COMMENT ON

1. Adding a comment for the current database

   ```sql
   COMMENT ON DATABASE IS 'It is a test (''my.fdb'') database';
   ```

2. Adding a comment for the METALS table

   ```sql
   COMMENT ON TABLE METALS IS 'Metal directory';
   ```

3. Adding a comment for the ISALLOY field in the METALS table

   ```sql
   COMMENT ON COLUMN METALS.ISALLOY IS '0 = fine metal, 1 = alloy';
   ```

4. Adding a comment for a parameter

   ```sql
   COMMENT ON PARAMETER ADD_EMP_PROJ.EMP_NO IS 'Employee ID';
   ```

5. Adding a comment for a package, its procedures and functions, and their parameters

   ```sql
   COMMENT ON PACKAGE APP_VAR IS 'Application Variables';
   COMMENT ON FUNCTION APP_VAR.GET_DATEBEGIN
   IS 'Returns the start date of the period';
   COMMENT ON PROCEDURE APP_VAR.SET_DATERANGE
   IS 'Set date range';
   COMMENT ON
   PROCEDURE PARAMETER APP_VAR.SET_DATERANGE.ADATEBEGIN
   ```
IS 'Start Date';
Chapter 6. Data Manipulation (DML) Statements

DML—data manipulation language—is the subset of SQL that is used by applications and procedural modules to extract and change data. Extraction, for the purpose of reading data, both raw and manipulated, is achieved with the SELECT statement. INSERT is for adding new data and DELETE is for erasing data that are no longer required. UPDATE, MERGE and UPDATE OR INSERT all modify data in various ways.

6.1. SELECT

**Used for**
Retrieving data

**Available in**
DSQL, ESQL, PSQL

**Global syntax**

```
[WITH [RECURSIVE] <cte> [, <cte> ...]]
SELECT
  [FIRST m] [SKIP n]
  [{DISTINCT | ALL}] <columns>
FROM
  <source> [[AS] alias]
  [<joins>]
[WHERE <condition>]
[GROUP BY <grouping-list>]
[HAVING <aggregate-condition>]]
[PLAN <plan-expr>]
[UNION [{DISTINCT | ALL}] <other-select>]
[ORDER BY <ordering-list>]
[ROWS <m> [TO <n>] |
  [OFFSET n {ROW | ROWS}]
  [FETCH {FIRST | NEXT} [m] {ROW | ROWS} ONLY]
}]
[FOR UPDATE [OF <columns>]]
[WITH LOCK]
[INTO <variables>]

<variables> ::= [:]varname [, [:]varname ...]
```

The SELECT statement retrieves data from the database and hands them to the application or the enclosing SQL statement. Data are returned in zero or more rows, each containing one or more columns or fields. The total of rows returned is the result set of the statement.

The only mandatory parts of the SELECT statement are:
• The SELECT keyword, followed by a columns list. This part specifies what you want to retrieve.
• The FROM keyword, followed by a selectable object. This tells the engine where you want to get it from.

In its most basic form, SELECT retrieves a number of columns from a single table or view, like this:

    select id, name, address
    from contacts

Or, to retrieve all the columns:

    select * from sales

In practice, a SELECT statement is usually executed with a WHERE clause, which limits the rows returned. The result set may be sorted by an ORDER BY clause, and FIRST … SKIP, OFFSET … FETCH or ROWS may further limit the number of returned rows, and can - for example - be used for pagination.

The column list may contain all kinds of expressions instead of just column names, and the source need not be a table or view: it may also be a derived table, a common table expression (CTE) or a selectable stored procedure (SP). Multiple sources may be combined in a JOIN, and multiple result sets may be combined in a UNION.

The following sections discuss the available SELECT subclauses and their usage in detail.

### 6.1.1. FIRST, SKIP

**Used for**
Retrieving a slice of rows from an ordered set

**Available in**
DSQL, PSQL

**Syntax**

```sql
SELECT
    [FIRST <m>] [SKIP <n>]
    FROM ...
...
<m>, <n> ::= ...
    <integer-literal>
| <query-parameter>
| (<integer-expression>)
```

*Table 68. Arguments for the FIRST and SKIP Clauses*
FIRST and SKIP are non-standard syntax

FIRST and SKIP are Firebird-specific clauses. Use the SQL-standard OFFSET, FETCH syntax wherever possible.

FIRST limits the output of a query to the first \( m \) rows. SKIP will suppress the given \( n \) rows before starting to return output.

FIRST and SKIP are both optional. When used together as in “FIRST \( m \) SKIP \( n \)”, the \( n \) topmost rows of the output set are discarded, and the first \( m \) rows of the rest of the set are returned.

Characteristics of FIRST and SKIP

- Any argument to FIRST and SKIP that is not an integer literal or an SQL parameter must be enclosed in parentheses. This implies that a subquery expression must be enclosed in two pairs of parentheses.
- SKIP 0 is allowed but totally pointless.
- FIRST 0 is also allowed and returns an empty set.
- Negative SKIP and/or FIRST values result in an error.
- If a SKIP lands past the end of the dataset, an empty set is returned.
- If the number of rows in the dataset (or the remainder left after a SKIP) is less than the value of the \( m \) argument supplied for FIRST, that smaller number of rows is returned. These are valid results, not error conditions.

Examples of FIRST/SKIP

1. The following query will return the first 10 names from the People table:

   ```sql
   select first 10 id, name from People
   order by name asc
   ```

2. The following query will return everything but the first 10 names:

   ```sql
   select skip 10 id, name from People
   order by name asc
   ```

3. And this one returns the last 10 rows. Notice the double parentheses:

   ```sql
   select skip ((select count(*) - 10 from People))
   ```
4. This query returns rows 81 to 100 of the People table:

```
select first 20 skip 80 id, name from People
order by name asc
```

See also
OFFSET, FETCH, ROWS

### 6.1.2. The SELECT Columns List

The columns list contains one or more comma-separated value expressions. Each expression provides a value for one output column. Alternatively, * ("select star" or "select all") can be used to stand for all the columns in a relation (i.e. a table, view or selectable stored procedure).

**Syntax**

```
SELECT
  [...]
  [{DISTINCT | ALL}] <output-column> [, <output-column> ...]
  [...]
  FROM ...

<output-column> ::= 
  { [[<qualifier>.]*/ 
  | <value-expression> [COLLATE collation] [[AS] alias] }

<value-expression> ::= 
  { [[<qualifier>.]table-column 
  | [[<qualifier>.]view-column 
  | [[<qualifier>.]selectable-SP-outparm 
  | <literal> 
  | <context-variable> 
  | <function-call> 
  | <single-value-subselect> 
  | <CASE-construct> 
  | any other expression returning a single 
  value of a Firebird data type or NULL }

<qualifier> ::= a relation name or alias
```

**Table 69. Arguments for the SELECT Columns List**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qualifier</td>
<td>Name of relation (view, stored procedure, derived table); or an alias for it</td>
</tr>
<tr>
<td>Argument</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>collation</td>
<td>Only for character-type columns: a collation name that exists and is valid for the character set of the data</td>
</tr>
<tr>
<td>alias</td>
<td>Column or field alias</td>
</tr>
<tr>
<td>table-column</td>
<td>Name of a table column</td>
</tr>
<tr>
<td>view-column</td>
<td>Name of a view column</td>
</tr>
<tr>
<td>selectable-SP-outparm</td>
<td>Declared name of an output parameter of a selectable stored procedure</td>
</tr>
<tr>
<td>constant</td>
<td>A constant</td>
</tr>
<tr>
<td>context-variable</td>
<td>Context variable</td>
</tr>
<tr>
<td>function-call</td>
<td>Scalar, aggregate, or window function expression</td>
</tr>
<tr>
<td>single-value-subselect</td>
<td>A subquery returning one scalar value (singleton)</td>
</tr>
<tr>
<td>CASE-construct</td>
<td>CASE construct setting conditions for a return value</td>
</tr>
<tr>
<td>other-single-value-expr</td>
<td>Any other expression returning a single value of a Firebird data type; or NULL</td>
</tr>
</tbody>
</table>

It is always valid to qualify a column name (or "**") with the name or alias of the table, view or selectable SP to which it belongs, followed by a dot (\'). For example, `relationname.columnname`, `relationname.*`, `alias.columnname`, `alias.*`. Qualifying is required if the column name occurs in more than one relation taking part in a join. Qualifying "**" is always mandatory if it is not the only item in the column list.

Warnings: Aliases hide the original relation name: once a table, view or procedure has been aliased, only the alias can be used as its qualifier throughout the query. The relation name itself becomes unavailable.

The column list may optionally be preceded by one of the keywords DISTINCT or ALL:

- **DISTINCT** filters out any duplicate rows. That is, if two or more rows have the same values in every corresponding column, only one of them is included in the result set
- **ALL** is the default: it returns all of the rows, including duplicates. ALL is rarely used; it is supported for compliance with the SQL standard.

A **COLLATE** clause will not change the appearance of the column as such. However, if the specified collation changes the case or accent sensitivity of the column, it may influence:

- The ordering, if an **ORDER BY** clause is also present and it involves that column
- Grouping, if the column is part of a **GROUP BY** clause
- The rows retrieved (and hence the total number of rows in the result set), if **DISTINCT** is used

**Examples of SELECT queries with different types of column lists**

A simple **SELECT** using only column names:
A query featuring a concatenation expression and a function call in the columns list:

```
select 'Mr./Mrs. ' || lastname, street, zip, upper(city)
from contacts
where date_last_purchase(id) = current_date
```

A query with two subselects:

```
select p.fullname,
    (select name from classes c where c.id = p.class) as class,
    (select name from mentors m where m.id = p.mentor) as mentor
from pupils p
```

The following query accomplishes the same as the previous one using joins instead of subselects:

```
select p.fullname,
    c.name as class,
    m.name as mentor
from pupils p
    join classes c on c.id = p.class
    join mentors m on m.id = p.mentor
```

This query uses a `CASE` construct to determine the correct title, e.g. when sending mail to a person:

```
select case upper(sex)
    when 'F' then 'Mrs.'
    when 'M' then 'Mr.'
    else ''
end as title,
    lastname,
    address
from employees
```

Query using a window function. Ranks employees by salary.

```
SELECT
    id,
    salary,
    name,
    DENSE_RANK() OVER (ORDER BY salary) AS EMP_RANK
```
FROM employees
ORDER BY salary;

Querying a selectable stored procedure:

```
select * from interesting_transactions(2010, 3, 'S')
order by amount
```

Selecting from columns of a derived table. A derived table is a parenthesized SELECT statement whose result set is used in an enclosing query as if it were a regular table or view. The derived table is shown in bold here:

```
select fieldcount,
count(relation) as num_tables
from (select r.rdb$relation_name as relation,
count(*) as fieldcount
from rdb$relations r
join rdb$relation_fields rf
on rf.rdb$relation_name = r.rdb$relation_name

group by relation)
group by fieldcount
```

Asking the time through a context variable (CURRENT_TIME):

```
select current_time from rdb$database
```

For those not familiar with RDB$DATABASE: this is a system table that is present in all Firebird databases and is guaranteed to contain exactly one row. Although it wasn’t created for this purpose, it has become standard practice among Firebird programmers to select from this table if you want to select “from nothing”, i.e. if you need data that are not bound to a table or view, but can be derived from the expressions in the output columns alone. Another example is:

```
select power(12, 2) as twelve_squared, power(12, 3) as twelve_cubed
from rdb$database
```

Finally, an example where you select some meaningful information from RDB$DATABASE itself:

```
select rdb$character_set_name from rdb$database
```

As you may have guessed, this will give you the default character set of the database.

See also
Functions, Aggregate Functions, Window Functions, Context Variables, CASE, Subqueries
6.1.3. The FROM clause

The FROM clause specifies the source(s) from which the data are to be retrieved. In its simplest form, this is just a single table or view. However, the source can also be a selectable stored procedure, a derived table, or a common table expression. Multiple sources can be combined using various types of joins.

This section focuses on single-source selects. Joins are discussed in a following section.

Syntax

```
SELECT ...
FROM <source>
[<joins>]
[...]
```

```
<source> ::= 
    table [[AS] alias] 
  | selectable-stored-procedure [[<args>]] [[AS] alias] 
  | <derived-table>
```

```
<derived-table> ::= 
    (<select-statement>) [[AS] alias] [[<column-aliases>]]
```

```
<column-aliases> ::= column-alias [, column-alias ...]
```

Table 70. Arguments for the FROM Clause

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>Name of a table, view or CTE</td>
</tr>
</tbody>
</table>
| selectable-stored-
  procedure              | Name of a selectable stored procedure                |
| args                    | Selectable stored procedure arguments                 |
| derived-table           | Derived table query expression                       |
| select-statement        | Any SELECT statement                                  |
| column-aliases          | Alias for a column in a relation, CTE or derived table|
| alias                   | The alias of a data source (table, view, procedure, CTE, derived table) |

Selecting FROM a table or view

When selecting from a single table or view, the FROM clause requires nothing more than the name. An alias may be useful or even necessary if there are subqueries that refer to the main select statement (as they often do — subqueries like this are called correlated subqueries).
Examples

```sql
select id, name, sex, age from actors
where state = 'Ohio'
```

```sql
select * from birds
where type = 'flightless'
order by family, genus, species
```

```sql
select firstname,
       middlename,
       lastname,
       date_of_birth,
       (select name from schools s where p.school = s.id) schoolname
from pupils p
where year_started = '2012'
order by schoolname, date_of_birth
```

Never mix column names with column aliases!

If you specify an alias for a table or a view, you must always use this alias in place of the table name whenever you query the columns of the relation (and wherever else you make a reference to columns, such as ORDER BY, GROUP BY and WHERE clauses).

Correct use:

```sql
SELECT PEARS
FROM FRUIT;
```

```sql
SELECT FRUIT.PEARS
FROM FRUIT;
```

```sql
SELECT PEARS
FROM FRUIT F;
```

```sql
SELECT F.PEARS
FROM FRUIT F;
```

Incorrect use:

```sql
SELECT FRUIT.PEARS
FROM FRUIT F;
```
Selecting FROM a stored procedure

A selectable stored procedure is a procedure that:

- contains at least one output parameter, and
- utilizes the SUSPEND keyword so the caller can fetch the output rows one by one, just as when selecting from a table or view.

The output parameters of a selectable stored procedure correspond to the columns of a regular table.

Selecting from a stored procedure without input parameters is just like selecting from a table or view:

```sql
select * from suspicious_transactions
  where assignee = 'John'
```

Any required input parameters must be specified after the procedure name, enclosed in parentheses:

```sql
select name, az, alt from visible_stars('Brugge', current_date, '22:30')
  where alt >= 20
  order by az, alt
```

Values for optional parameters (that is, parameters for which default values have been defined) may be omitted or provided. However, if you provide them only partly, the parameters you omit must all be at the tail end.

Supposing that the procedure `visible_stars` from the previous example has two optional parameters: `min_magn` (numeric(3,1)) and `spectral_class` (varchar(12)), the following queries are all valid:

```sql
select name, az, alt
  from visible_stars('Brugge', current_date, '22:30');

select name, az, alt
  from visible_stars('Brugge', current_date, '22:30', 4.0);

select name, az, alt
  from visible_stars('Brugge', current_date, '22:30', 4.0, 'G');
```

But this one isn’t, because there’s a “hole” in the parameter list:

```sql
select name, az, alt
  from visible_stars('Brugge', current_date, '22:30', 'G');
```
An alias for a selectable stored procedure is specified after the parameter list:

```
select
    number,
    (select name from contestants c where c.number = gw.number)
from get_winners('#34517', 'AMS') gw
```

If you refer to an output parameter ("column") by qualifying it with the full procedure name, the procedure alias should be omitted:

```
select
    number,
    (select name from contestants c where c.number = get_winners.number)
from get_winners('#34517', 'AMS')
```

See also

*Stored Procedures, CREATE PROCEDURE*

**Selecting FROM a derived table**

A derived table is a valid SELECT statement enclosed in parentheses, optionally followed by a table alias and/or column aliases. The result set of the statement acts as a virtual table which the enclosing statement can query.

**Syntax**

```
(<select-query>)
    [[AS] derived-table-alias]
    [((<derived-column-aliases>))]
    <derived-column-aliases> := column-alias [, column-alias ...]
```

The set returned data set by this "SELECT FROM (SELECT FROM..)" style of statement is a virtual table that can be queried within the enclosing statement, as if it were a regular table or view.

**Example using a derived table**

The derived table in the query below returns the list of table names in the database, and the number of columns in each table. A "drill-down" query on the derived table returns the counts of fields and the counts of tables having each field count:

```
SELECT
    FIELDCOUNT,
    COUNT(RELATION) AS NUM_TABLES
FROM (SELECT
    R.RDB$RELATION_NAME RELATION,
    COUNT(*) AS FIELDCOUNT
```
FROM RDB$RELATIONS R
JOIN RDB$RELATION_FIELDS RF
ON RF.RDB$RELATION_NAME = R.RDB$RELATION_NAME
GROUP BY RELATION)
GROUP BY FIELDCOUNT

A trivial example demonstrating how the alias of a derived table and the list of column aliases (both optional) can be used:

SELECT
  DBINFO.DESCR, DBINFO.DEF_CHARSET
FROM (SELECT *
       FROM RDB$DATABASE) DBINFO
  (DESCR, REL_ID, SEC_CLASS, DEF_CHARSET)

More about Derived Tables

Derived tables can

- be nested
- be unions, and can be used in unions
- contain aggregate functions, subqueries and joins
- be used in aggregate functions, subqueries and joins
- be calls to selectable stored procedures or queries to them
- have WHERE, ORDER BY and GROUP BY clauses, FIRST/SKIP or ROWS directives, et al.

Furthermore,

- Each column in a derived table must have a name. If it does not have a name, such as when it is a constant or a run-time expression, it should be given an alias, either in the regular way or by including it in the list of column aliases in the derived table’s specification.
  - The list of column aliases is optional but, if it exists, it must contain an alias for every column in the derived table
- The optimizer can process derived tables very effectively. However, if a derived table is included in an inner join and contains a subquery, the optimizer will be unable to use any join order.

A more useful example

Suppose we have a table COEFFS which contains the coefficients of a number of quadratic equations we have to solve. It has been defined like this:

```
create table coeffs (  
a double precision not null,
```
Depending on the values of \(a\), \(b\) and \(c\), each equation may have zero, one or two solutions. It is possible to find these solutions with a single-level query on table `COEFFS`, but the code will look rather messy and several values (like the discriminant) will have to be calculated multiple times per row. A derived table can help keep things clean here:

```sql
select
  iif (D >= 0, (-b - sqrt(D)) / denom, null) sol_1,
  iif (D > 0, (-b + sqrt(D)) / denom, null) sol_2
from
  (select b, b*b - 4*a*c, 2*a from coeffs) (b, D, denom)
```

If we want to show the coefficients next to the solutions (which may not be a bad idea), we can alter the query like this:

```sql
select
  a, b, c,
  iif (D >= 0, (-b - sqrt(D)) / denom, null) sol_1,
  iif (D > 0, (-b + sqrt(D)) / denom, null) sol_2
from
  (select a, b, c, b*b - 4*a*c as D, 2*a as denom
   from coeffs)
```

Notice that whereas the first query used a column aliases list for the derived table, the second adds aliases internally where needed. Both methods work, as long as every column is guaranteed to have a name.

All columns in the derived table will be evaluated as many times as they are specified in the main query. This is important, as it can lead to unexpected results when using non-deterministic functions. The following shows an example of this.

```sql
SELECT
  UUID_TO_CHAR(X) AS C1,
  UUID_TO_CHAR(X) AS C2,
  UUID_TO_CHAR(X) AS C3
FROM (SELECT GEN_UUID() AS X
      FROM RDB$DATABASE) T;
```

The result if this query produces three different values:

```
C1  80AAECE6-65CD-4C2F-90A8-5D541C3C7279
C2  C1214CD3-423C-406D-B5BD-95BF432ED3E3
```
To ensure a single result of the `GEN_UUID` function, you can use the following method:

```sql
SELECT
    UUID_TO_CHAR(X) AS C1,
    UUID_TO_CHAR(X) AS C2,
    UUID_TO_CHAR(X) AS C3
FROM (SELECT GEN_UUID() AS X
    FROM RDB$DATABASE
    UNION ALL
    SELECT NULL FROM RDB$DATABASE WHERE 1 = 0) T;
```

This query produces a single result for all three columns:

```
C1  80AAECED-65CD-4C2F-90AB-5D548C3C7279
C2  80AAECED-65CD-4C2F-90AB-5D548C3C7279
C3  80AAECED-65CD-4C2F-90AB-5D548C3C7279
```

An alternative solution is to wrap the `GEN_UUID` query in a subquery:

```sql
SELECT
    UUID_TO_CHAR(X) AS C1,
    UUID_TO_CHAR(X) AS C2,
    UUID_TO_CHAR(X) AS C3
FROM (SELECT
    (SELECT GEN_UUID() FROM RDB$DATABASE) AS X
    FROM RDB$DATABASE) T;
```

This is an artifact of the current implementation. This behaviour may change in a future Firebird version.

**Selecting FROM a Common Table Expression (CTE)**

A common table expression — or CTE — is a more complex variant of the derived table, but it is also more powerful. A preamble, starting with the keyword `WITH`, defines one or more named CTE's, each with an optional column aliases list. The main query, which follows the preamble, can then access these CTE's as if they were regular tables or views. The CTE's go out of scope once the main query has run to completion.

For a full discussion of CTE's, please refer to the section Common Table Expressions ("WITH ... AS ... SELECT").

The following is a rewrite of our derived table example as a CTE:
with vars (b, D, denom) as (  
    select b, b*b - 4*a*c, 2*a from coeffs  
)
select  
  iif (D >= 0, (-b - sqrt(D)) / denom, null) sol_1,  
  iif (D > 0, (-b + sqrt(D)) / denom, null) sol_2  
from vars

Except for the fact that the calculations that have to be made first are now at the beginning, this isn't a great improvement over the derived table version. However, we can now also eliminate the double calculation of \( \sqrt{D} \) for every row:

with vars (b, D, denom) as (  
    select b, b*b - 4*a*c, 2*a from coeffs  
),
vars2 (b, D, denom, sqrtD) as (  
    select b, D, denom, iif (D >= 0, sqrt(D), null) from vars  
)
select  
  iif (D >= 0, (-b - sqrtD) / denom, null) sol_1,  
  iif (D > 0, (-b + sqrtD) / denom, null) sol_2  
from vars2

The code is a little more complicated now, but it might execute more efficiently (depending on what takes more time: executing the \( \text{SQRT} \) function or passing the values of \( b, D \) and \( \text{denom} \) through an extra \( \text{CTE} \)). Incidentally, we could have done the same with derived tables, but that would involve nesting.

All columns in the CTE will be evaluated as many times as they are specified in the main query. This is important, as it can lead to unexpected results when using non-deterministic functions. The following shows an example of this.

WITH T (X) AS (  
    SELECT GEN_UUID()  
    FROM RDB$DATABASE)  
SELECT  
  UUID_TO_CHAR(X) as c1,  
  UUID_TO_CHAR(X) as c2,  
  UUID_TO_CHAR(X) as c3  
FROM T

The result if this query produces three different values:

| C1  | 80AAECE6D-65CD-4C2F-90AB-5D548C3C7279 |
| C2  | C1214CD3-423C-406D-B5BD-95BF432ED3E3 |
To ensure a single result of the `GEN_UUID` function, you can use the following method:

```sql
WITH T (X) AS (
    SELECT GEN_UUID()
    FROM RDB$DATABASE
    UNION ALL
    SELECT NULL FROM RDB$DATABASE WHERE 1 = 0)
SELECT
    UUID_TO_CHAR(X) as c1,
    UUID_TO_CHAR(X) as c2,
    UUID_TO_CHAR(X) as c3
FROM T;
```

This query produces a single result for all three columns:

```
C1  80AAECED-65CD-4C2F-90AB-5D548C3C7279
C2  80AAECED-65CD-4C2F-90AB-5D548C3C7279
C3  80AAECED-65CD-4C2F-90AB-5D548C3C7279
```

An alternative solution is to wrap the `GEN_UUID` query in a subquery:

```sql
WITH T (X) AS (
    SELECT (SELECT GEN_UUID() FROM RDB$DATABASE)
    FROM RDB$DATABASE)
SELECT
    UUID_TO_CHAR(X) as c1,
    UUID_TO_CHAR(X) as c2,
    UUID_TO_CHAR(X) as c3
FROM T;
```

This is an artifact of the current implementation. This behaviour may change in a future Firebird version.

See also

Common Table Expressions ("WITH … AS … SELECT").

### 6.1.4. Joins

Joins combine data from two sources into a single set. This is done on a row-by-row basis and usually involves checking a join condition in order to determine which rows should be merged and appear in the resulting dataset. There are several types (INNER, OUTER) and classes (qualified, natural, etc.) of joins, each with its own syntax and rules.
Since joins can be chained, the datasets involved in a join may themselves be joined sets.

Syntax

```
SELECT ...
FROM <source> [<joins>]
[...]
```

```
<source> ::= table [[AS] alias] |
selectable-stored-procedure [[(<args>)]] [[AS] alias] |
<derived-table>

<joins> ::= <join> [<join> ...]

<join> ::= [<join-type>] JOIN <source> <join-condition> |
NATURAL [INNER | {LEFT | RIGHT | FULL} [OUTER] <source> |
{CROSS JOIN | ,} <source>

<join-type> ::= INNER | {LEFT | RIGHT | FULL} [OUTER]
<join-condition> ::= ON <condition> | USING (<column-list>)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>Name of a table, view or CTE</td>
</tr>
<tr>
<td>selectable-stored-procedure</td>
<td>Name of a selectable stored procedure</td>
</tr>
<tr>
<td>args</td>
<td>Selectable stored procedure input parameter(s)</td>
</tr>
<tr>
<td>derived-table</td>
<td>Derived table query expression</td>
</tr>
<tr>
<td>alias</td>
<td>An alias for a data source (table, view, procedure, CTE, derived table)</td>
</tr>
<tr>
<td>condition</td>
<td>Join condition (criterion)</td>
</tr>
<tr>
<td>column-list</td>
<td>The list of columns used for an equi-join</td>
</tr>
</tbody>
</table>

**Inner vs. Outer Joins**

A join always combines data rows from two sets (usually referred to as the left set and the right set). By default, only rows that meet the join condition (i.e. that match at least one row in the other set when the join condition is applied) make it into the result set. This default type of join is called an inner join. Suppose we have the following two tables:

**Table A**
If we join these tables like this:

```sql
select *
from A
join B on A.id = B.code;
```

then the result set will be:

<table>
<thead>
<tr>
<th>ID</th>
<th>S</th>
<th>CODE</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>Just some text</td>
<td>87</td>
<td>416.0</td>
</tr>
<tr>
<td>235</td>
<td>Silence</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first row of A has been joined with the second row of B because together they met the condition “A.id = B.code”. The other rows from the source tables have no match in the opposite set and are therefore not included in the join. Remember, this is an INNER join. We can make that fact explicit by writing:

```sql
select *
from A
inner join B on A.id = B.code;
```

However, since INNER is the default, it is usually ommitted.

It is perfectly possible that a row in the left set matches several rows from the right set or vice versa. In that case, all those combinations are included, and we can get results like:

<table>
<thead>
<tr>
<th>ID</th>
<th>S</th>
<th>CODE</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>Just some text</td>
<td>87</td>
<td>416.0</td>
</tr>
<tr>
<td>87</td>
<td>Just some text</td>
<td>87</td>
<td>-1.0</td>
</tr>
<tr>
<td>-23</td>
<td>Don’t know</td>
<td>-23</td>
<td>56.7735</td>
</tr>
<tr>
<td>-23</td>
<td>Still don’t know</td>
<td>-23</td>
<td>56.7735</td>
</tr>
<tr>
<td>-23</td>
<td>I give up</td>
<td>-23</td>
<td>56.7735</td>
</tr>
</tbody>
</table>

Sometimes we want (or need) all the rows of one or both of the sources to appear in the joined set,
regardless of whether they match a record in the other source. This is where outer joins come in. A LEFT outer join includes all the records from the left set, but only matching records from the right set. In a RIGHT outer join it’s the other way around. FULL outer joins include all the records from both sets. In all outer joins, the “holes” (the places where an included source record doesn’t have a match in the other set) are filled up with NULLs.

In order to make an outer join, you must specify LEFT, RIGHT or FULL, optionally followed by the keyword OUTER.

Below are the results of the various outer joins when applied to our original tables A and B:

```sql
select * 
from A
left [outer] join B on A.id = B.code;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>S</th>
<th>CODE</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>Just some text</td>
<td>87</td>
<td>416.0</td>
</tr>
<tr>
<td>235</td>
<td>Silence</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

```sql
select * 
from A
right [outer] join B on A.id = B.code
```

<table>
<thead>
<tr>
<th>ID</th>
<th>S</th>
<th>CODE</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
<td>-23</td>
<td>56.7735</td>
</tr>
<tr>
<td>87</td>
<td>Just some text</td>
<td>87</td>
<td>416.0</td>
</tr>
</tbody>
</table>

```sql
select * 
from A
full [outer] join B on A.id = B.code
```

<table>
<thead>
<tr>
<th>ID</th>
<th>S</th>
<th>CODE</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
<td>-23</td>
<td>56.7735</td>
</tr>
<tr>
<td>87</td>
<td>Just some text</td>
<td>87</td>
<td>416.0</td>
</tr>
<tr>
<td>235</td>
<td>Silence</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

**Qualified joins**

Qualified joins specify conditions for the combining of rows. This happens either explicitly in a ON clause or implicitly in a USING clause.
Explicit-condition joins

Most qualified joins have an \texttt{ON} clause, with an explicit condition that can be any valid Boolean expression, but usually involves some comparison between the two sources involved.

Quite often, the condition is an equality test (or a number of ANDed equality tests) using the \(\texttt{=}\) operator. Joins like these are called \textit{equi-joins}. (The examples in the section on inner and outer joins were all equi-joins.)

Examples of joins with an explicit condition:

```sql
/* Select all Detroit customers who made a purchase in 2013, along with the purchase details: */
select * from customers c
    join sales s on s.cust_id = c.id
    where c.city = 'Detroit' and s.year = 2013;

/* Same as above, but include non-buying customers: */
select * from customers c
    left join sales s on s.cust_id = c.id
    where c.city = 'Detroit' and s.year = 2013;

/* For each man, select the women who are taller than he. Men for whom no such woman exists are not included. */
select m.fullname as man, f.fullname as woman
    from males m
    join females f on f.height > m.height;

/* Select all pupils with their class and mentor. Pupils without a mentor are also included. Pupils without a class are not included. */
select p.firstname, p.middlename, p.lastname, c.name, m.name
    from pupils p
    join classes c on c.id = p.class
    left join mentors m on m.id = p.mentor;
```
Named columns joins

Equi-joins often compare columns that have the same name in both tables. If this is the case, we can also use the second type of qualified join: the named columns join.

Named columns joins are not supported in Dialect 1 databases.

Named columns joins have a USING clause which states just the column names. So instead of this:

```sql
select * from flotsam f
join jetsam j
on f.sea = j.sea
and f.ship = j.ship;
```

we can also write:

```sql
select * from flotsam
join jetsam using (sea, ship)
```

which is considerably shorter. The result set is a little different though—at least when using “SELECT *”:

- The explicit-condition join — with the ON clause — will contain each of the columns SEA and SHIP twice: once from table FLOTSAM, and once from table JETSAM. Obviously, they will have the same values.
- The named columns join — with the USING clause — will contain these columns only once.

If you want all the columns in the result set of the named columns join, set up your query like this:

```sql
select f.*, j.*
from flotsam f
join jetsam j using (sea, ship);
```

This will give you the exact same result set as the explicit-condition join.

For an OUTER named columns join, there’s an additional twist when using “SELECT *” or an unqualified column name from the USING list:

If a row from one source set doesn’t have a match in the other but must still be included because of the LEFT, RIGHT or FULL directive, the merged column in the joined set gets the non-NULL value. That is fair enough, but now you can’t tell whether this value came from the left set, the right set, or both. This can be especially deceiving when the value came from the right hand set, because “*” always shows combined columns in the left hand part—even in the case of a RIGHT join.

Whether this is a problem or not depends on the situation. If it is, use the “a.*, b.*” approach shown above, with a and b the names or aliases of the two sources. Or better yet, avoid “*” altogether in your serious queries and qualify all column names in joined sets. This has the
additional benefit that it forces you to think about which data you want to retrieve and where from.

It is your responsibility to make sure the column names in the USING list are of compatible types between the two sources. If the types are compatible but not equal, the engine converts them to the type with the broadest range of values before comparing the values. This will also be the data type of the merged column that shows up in the result set if “SELECT *” or the unqualified column name is used. Qualified columns on the other hand will always retain their original data type.

If, when joining by named columns, you are using a join column in the WHERE clause, always use the qualified column name, otherwise an index on this column will not be used.

```
SELECT 1 FROM t1 a JOIN t2 b USING (x) WHERE x = 0;
-- PLAN JOIN (A NATURAL, B INDEX (RDB$2))
```

However:

```
SELECT 1 FROM t1 a JOIN t2 b USING (x) WHERE a.x = 0;
-- PLAN JOIN (A INDEX (RDB$1), B INDEX (RDB$2))
```

```
SELECT 1 FROM t1 a JOIN t2 b USING (x) WHERE b.x = 0;
-- PLAN JOIN (A INDEX (RDB$1), B INDEX (RDB$2))
```

The fact is, the unspecified column in this case is implicitly replaced by `COALESCE(a.x, b.x)`. This clever trick is used to disambiguate column names, but it also interferes with the use of the index.

### Natural joins

Taking the idea of the named columns join a step further, a natural join performs an automatic equi-join on all the columns that have the same name in the left and right table. The data types of these columns must be compatible.

Natural joins are not supported in Dialect 1 databases.

### Syntax

```
<natural-join> ::= NATURAL [<join-type>] JOIN <source>

<join-type> ::= INNER | {LEFT | RIGHT | FULL} [OUTER]
```

Given these two tables:

```
create table TA (a bigint,
```
create table TA (  
a bigint,  
descr varchar(12),  
x float,  
is_date date  
);

create table TB (  
a bigint,  
descr varchar(12),  
x float,  
is_date date  
);

A natural join on TA and TB would involve the columns a and ins_date, and the following two statements would have the same effect:

```
select * from TA  
natural join TB;
```

```
select * from TA  
join TB using (a, ins_date);
```

Like all joins, natural joins are inner joins by default, but you can turn them into outer joins by specifying LEFT, RIGHT or FULL before the JOIN keyword.

If there are no columns with the same name in the two source relations, a CROSS JOIN is performed. We'll get to this type of join in a minute.

**Cross joins**

A cross join produces the full set product of the two data sources. This means that it successfully matches every row in the left source to every row in the right source.

**Syntax**

```
<cross-join> ::= {CROSS JOIN | ,} <source>
```

Use of the comma syntax is discouraged, and we recommend using the explicit join syntax.

Cross-joining two sets is equivalent to joining them on a tautology (a condition that is always true). The following two statements have the same effect:

```
select * from TA  
cross join TB;
```

```
select * from TA
```
Cross joins are inner joins, because they only include matching records – it just so happens that every record matches! An outer cross join, if it existed, wouldn't add anything to the result, because what outer joins add are non-matching records, and these don't exist in cross joins.

Cross joins are seldom useful, except if you want to list all the possible combinations of two or more variables. Suppose you are selling a product that comes in different sizes, different colors and different materials. If these variables are each listed in a table of their own, this query would return all the combinations:

```sql
select m.name, s.size, c.name
from materials m
cross join sizes s
cross join colors c;
```

### Implicit Joins

In the SQL:89 standard, the tables involved in a join were specified as a comma-delimited list in the `FROM` clause (in other words, a cross join). The join conditions were then specified in the `WHERE` clause among other search terms. This type of join is called an implicit join.

An example of an implicit join:

```sql
/*
* A sample of all Detroit customers who
* made a purchase.
*/
SELECT *
FROM customers c, sales s
WHERE s.cust_id = c.id AND c.city = 'Detroit'
```

### Mixing Explicit and Implicit Joins

Mixing explicit and implicit joins is not recommend, but is allowed. However, some types of mixing are not supported by Firebird.

For example, the following query will raise the error “Column does not belong to referenced table”

```sql
SELECT *
FROM TA, TB
JOIN TC ON TA.COL1 = TC.COL1
WHERE TA.COL2 = TB.COL2
```

That is because the explicit join cannot see the TA table. However, the next query will complete without error, since the restriction is not violated.
A Note on Equality

This note about equality and inequality operators applies everywhere in Firebird’s SQL language, not just in JOIN conditions.

The “=” operator, which is explicitly used in many conditional joins and implicitly in named column joins and natural joins, only matches values to values. According to the SQL standard, "NULL" is not a value and hence two NULLs are neither equal nor unequal to one another. If you need NULLs to match each other in a join, use the IS NOT DISTINCT FROM operator. This operator returns true if the operands have the same value or if they are both NULL.

```sql
select * 
from A join B 
on A.id is not distinct from B.code;
```

Likewise, in the—extremely rare—cases where you want to join on inequality, use IS DISTINCT FROM, not “<>”, if you want NULL to be considered different from any value and two NULLs considered equal:

```sql
select * 
from A join B 
on A.id is distinct from B.code;
```

Ambiguous field names in joins

Firebird rejects unqualified field names in a query if these field names exist in more than one dataset involved in a join. This is even true for inner equi-joins where the field name figures in the ON clause like this:

```sql
select a, b, c 
from TA 
join TB on TA.a = TB.a;
```

There is one exception to this rule: with named columns joins and natural joins, the unqualified field name of a column taking part in the matching process may be used legally and refers to the merged column of the same name. For named columns joins, these are the columns listed in the USING clause. For natural joins, they are the columns that have the same name in both relations. But please notice again that, especially in outer joins, plain colname isn’t always the same as left.colname or right.colname. Types may differ, and one of the qualified columns may be NULL.
while the other isn’t. In that case, the value in the merged, unqualified column may mask the fact that one of the source values is absent.

6.1.5. The **WHERE** clause

The **WHERE** clause serves to limit the rows returned to the ones that the caller is interested in. The condition following the keyword **WHERE** can be as simple as a check like “`AMOUNT = 3`” or it can be a multilayered, convoluted expression containing subselects, predicates, function calls, mathematical and logical operators, context variables and more.

The condition in the **WHERE** clause is often called the *search condition*, the *search expression* or simply the *search*.

In DSQL and ESQL, the search expression may contain parameters. This is useful if a query has to be repeated a number of times with different input values. In the SQL string as it is passed to the server, question marks are used as placeholders for the parameters. They are called *positional parameters* because they can only be told apart by their position in the string. Connectivity libraries often support *named parameters* of the form `:id`, `:amount`, `:a` etc. These are more user-friendly; the library takes care of translating the named parameters to positional parameters before passing the statement to the server.

The search condition may also contain local (PSQL) or host (ESQL) variable names, preceded by a colon.

**Syntax**

```
SELECT ...  
FROM ...  
[...]  
WHERE <search-condition>  
[...]
```

**Table 72. Argument of **WHERE**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>search-condition</td>
<td>A Boolean expression returning TRUE, FALSE or possibly UNKNOWN (NULL)</td>
</tr>
</tbody>
</table>

Only those rows for which the search condition evaluates to TRUE are included in the result set. Be careful with possible NULL outcomes: if you negate a NULL expression with **NOT**, the result will still be NULL and the row will not pass. This is demonstrated in one of the examples below.

**Examples**

```
select genus, species from mammals  
  where family = 'Felidae'  
order by genus;
```
The following example shows what can happen if the search condition evaluates to NULL.

Suppose you have a table listing some children’s names and the number of marbles they possess. At a certain moment, the table contains these data:

<table>
<thead>
<tr>
<th>CHILD</th>
<th>MARBLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anita</td>
<td>23</td>
</tr>
<tr>
<td>Bob E.</td>
<td>12</td>
</tr>
<tr>
<td>Chris</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>Deirdre</td>
<td>1</td>
</tr>
<tr>
<td>Eve</td>
<td>17</td>
</tr>
<tr>
<td>Fritz</td>
<td>0</td>
</tr>
</tbody>
</table>
First, please notice the difference between NULL and 0: Fritz is *known* to have no marbles at all, Chris's and Hadassah's marble counts are unknown.

Now, if you issue this SQL statement:

```sql
select list(child) from marbletable where marbles > 10;
```

you will get the names Anita, Bob E., Eve and Gerry. These children all have more than 10 marbles.

If you negate the expression:

```sql
select list(child) from marbletable where not marbles > 10
```

it’s the turn of Deirdre, Fritz and Isaac to fill the list. Chris and Hadassah are not included, because they aren’t *known* to have ten marbles or less. Should you change that last query to:

```sql
select list(child) from marbletable where marbles <= 10;
```

the result will still be the same, because the expression NULL <= 10 yields UNKNOWN. This is not the same as TRUE, so Chris and Hadassah are not listed. If you want them listed with the “poor” children, change the query to:

```sql
select list(child) from marbletable
    where marbles <= 10 or marbles is null;
```

Now the search condition becomes true for Chris and Hadassah, because “marbles is null” obviously returns TRUE in their case. In fact, the search condition cannot be NULL for anybody now.

Lastly, two examples of SELECT queries with parameters in the search. It depends on the application how you should define query parameters and even if it is possible at all. Notice that queries like these cannot be executed immediately: they have to be *prepared* first. Once a parameterized query has been prepared, the user (or calling code) can supply values for the parameters and have it executed many times, entering new values before every call. How the values are entered and the execution started is up to the application. In a GUI environment, the user typically types the parameter values in one or more text boxes and then clicks an “Execute”, “Run” or “Refresh” button.

```sql
select name, address, phone from stores
```
The last query cannot be passed directly to the engine; the application must convert it to the other format first, mapping named parameters to positional parameters.

6.1.6. The **GROUP BY** clause

**GROUP BY** merges output rows that have the same combination of values in its item list into a single row. Aggregate functions in the select list are applied to each group individually instead of to the dataset as a whole.

If the select list only contains aggregate columns or, more generally, columns whose values don’t depend on individual rows in the underlying set, **GROUP BY** is optional. When omitted, the final result set of will consist of a single row (provided that at least one aggregated column is present).

If the select list contains both aggregate columns and columns whose values may vary per row, the **GROUP BY** clause becomes mandatory.

**Syntax**

```sql
SELECT ... FROM ...
    GROUP BY <grouping-item> [, <grouping-item> ...]
[HAVING <grouped-row-condition>]
...
```

**Arguments for the GROUP BY Clause**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-aggr-expression</td>
<td>Any non-aggregating expression that is not included in the SELECT list, i.e. unselected columns from the source set or expressions that do not depend on the data in the set at all</td>
</tr>
<tr>
<td>column-copy</td>
<td>A literal copy, from the SELECT list, of an expression that contains no aggregate function</td>
</tr>
<tr>
<td>column-alias</td>
<td>The alias, from the SELECT list, of an expression (column) that contains no aggregate function</td>
</tr>
</tbody>
</table>
A general rule of thumb is that every non-aggregate item in the SELECT list must also be in the GROUP BY list. You can do this in three ways:

1. By copying the item verbatim from the select list, e.g. “class” or “’D:’ || upper(doccode)”.
2. By specifying the column alias, if it exists.
3. By specifying the column position as an integer literal between 1 and the number of columns. Integer values resulting from expressions or parameter substitutions are simply invariables and will be used as such in the grouping. They will have no effect though, as their value is the same for each row.

If you group by a column position, the expression at that position is copied internally from the select list. If it concerns a subquery, that subquery will be executed again in the grouping phase. That is to say, grouping by the column position, rather than duplicating the subquery expression in the grouping clause, saves keystrokes and bytes, but it is not a way of saving processing cycles!

In addition to the required items, the grouping list may also contain:

- Columns from the source table that are not in the select list, or non-aggregate expressions based on such columns. Adding such columns may further subdivide the groups. However, since these columns are not in the select list, you can't tell which aggregated row corresponds to which value in the column. So, in general, if you are interested in this information, you also include the column or expression in the select list—which brings you back to the rule: “every non-aggregate column in the select list must also be in the grouping list”.

- Expressions that aren't dependent on the data in the underlying set, e.g. constants, context variables, single-value non-correlated subselects etc. This is only mentioned for completeness, as adding such items is utterly pointless: they don't affect the grouping at all. “Harmless but useless” items like these may also figure in the select list without being copied to the grouping list.

**Examples**

When the select list contains only aggregate columns, GROUP BY is not mandatory:

```sql
select count(*), avg(age) from students
where sex = 'M';
```

This will return a single row listing the number of male students and their average age. Adding expressions that don't depend on values in individual rows of table STUDENTS doesn't change that:

```sql
select count(*), avg(age), current_date from students
```
where sex = 'M';

The row will now have an extra column showing the current date, but other than that, nothing fundamental has changed. A GROUP BY clause is still not required.

However, in both the above examples it is allowed. This is perfectly valid:

```sql
select count(*), avg(age) from students
  where sex = 'M'
  group by class;
```

This will return a row for each class that has boys in it, listing the number of boys and their average age in that particular class. (If you also leave the current_date field in, this value will be repeated on every row, which is not very exciting.)

The above query has a major drawback though: it gives you information about the different classes, but it doesn’t tell you which row applies to which class. In order to get that extra bit of information, the non-aggregate column CLASS must be added to the select list:

```sql
select class, count(*), avg(age) from students
  where sex = 'M'
  group by class;
```

Now we have a useful query. Notice that the addition of column CLASS also makes the GROUP BY clause mandatory. We can’t drop that clause anymore, unless we also remove CLASS from the column list.

The output of our last query may look something like this:

<table>
<thead>
<tr>
<th>CLASS</th>
<th>COUNT</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>2A</td>
<td>12</td>
<td>13.5</td>
</tr>
<tr>
<td>2B</td>
<td>9</td>
<td>13.9</td>
</tr>
<tr>
<td>3A</td>
<td>11</td>
<td>14.6</td>
</tr>
<tr>
<td>3B</td>
<td>12</td>
<td>14.4</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

The headings “COUNT” and “AVG” are not very informative. In a simple case like this, you might get away with that, but in general you should give aggregate columns a meaningful name by aliasing them:

```sql
select class,
  count(*) as num_boys,
  avg(age) as boys_avg_age
from students
where sex = 'M'
```
As you may recall from the formal syntax of the columns list, the `AS` keyword is optional.

Adding more non-aggregate (or rather: row-dependent) columns requires adding them to the `GROUP BY` clause too. For instance, you might want to see the above information for girls as well; and you may also want to differentiate between boarding and day students:

```sql
select class, sex, boarding_type, count(*) as number, avg(age) as avg_age
from students
group by class, sex, boarding_type;
```

This may give you the following result:

<table>
<thead>
<tr>
<th>CLASS</th>
<th>SEX</th>
<th>BOARDING_TYPE</th>
<th>NUMBER</th>
<th>AVG_AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2A</td>
<td>F</td>
<td>BOARDING</td>
<td>9</td>
<td>13.3</td>
</tr>
<tr>
<td>2A</td>
<td>F</td>
<td>DAY</td>
<td>6</td>
<td>13.5</td>
</tr>
<tr>
<td>2A</td>
<td>M</td>
<td>BOARDING</td>
<td>7</td>
<td>13.6</td>
</tr>
<tr>
<td>2A</td>
<td>M</td>
<td>DAY</td>
<td>5</td>
<td>13.4</td>
</tr>
<tr>
<td>2B</td>
<td>F</td>
<td>BOARDING</td>
<td>11</td>
<td>13.7</td>
</tr>
<tr>
<td>2B</td>
<td>F</td>
<td>DAY</td>
<td>5</td>
<td>13.7</td>
</tr>
<tr>
<td>2B</td>
<td>M</td>
<td>BOARDING</td>
<td>6</td>
<td>13.8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Each row in the result set corresponds to one particular combination of the columns `CLASS`, `SEX` and `BOARDING_TYPE`. The aggregate results—number and average age—are given for each of these rather specific groups individually. In a query like this, you don’t see a total for boys as a whole, or day students as a whole. That’s the tradeoff: the more non-aggregate columns you add, the more you can pinpoint very specific groups, but the more you also lose sight of the general picture. Of course, you can still obtain the “coarser” aggregates through separate queries.

**HAVING**

Just as a `WHERE` clause limits the rows in a dataset to those that meet the search condition, so the `HAVING` sub-clause imposes restrictions on the aggregated rows in a grouped set. `HAVING` is optional, and can only be used in conjunction with `GROUP BY`.

The condition(s) in the `HAVING` clause can refer to:

- Any aggregated column in the select list. This is the most widely used case.
- Any aggregated expression that is not in the select list, but allowed in the context of the query.
This is sometimes useful too.

- Any column in the `GROUP BY` list. While legal, it is more efficient to filter on these non-aggregated data at an earlier stage: in the `WHERE` clause.
- Any expression whose value doesn’t depend on the contents of the dataset (like a constant or a context variable). This is valid but utterly pointless, because it will either suppress the entire set or leave it untouched, based on conditions that have nothing to do with the set itself.

A `HAVING` clause can *not* contain:

- Non-aggregated column expressions that are not in the `GROUP BY` list.
- Column positions. An integer in the `HAVING` clause is just an integer.
- Column aliases — not even if they appear in the `GROUP BY` clause!

**Examples**

Building on our earlier examples, this could be used to skip small groups of students:

```sql
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
from students
where sex = 'M'
group by class
having count(*) >= 5;
```

To select only groups that have a minimum age spread:

```sql
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
from students
where sex = 'M'
group by class
having max(age) - min(age) > 1.2;
```

Notice that if you’re really interested in this information, you’d normally include `min(age)` and `max(age)` — or the expression “`max(age) - min(age)`” — in the select list as well!

To include only 3rd classes:

```sql
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
from students
where sex = 'M'
group by class
```
Better would be to move this condition to the `WHERE` clause:

```sql
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
from students
where sex = 'M' and class starting with '3'
group by class;
```

### 6.1.7. The PLAN clause

The PLAN clause enables the user to submit a data retrieval plan, thus overriding the plan that the optimizer would have generated automatically.

**Syntax**

```
PLAN <plan-expr>

<plan-expr> ::= 
    (<plan-item> [, <plan-item> ...])
    | <sorted-item>
    | <joined-item>
    | <merged-item>
    | <hash-item>

<sorted-item> ::= SORT (<plan-item>)

<joined-item> ::= 
    JOIN (<plan-item>, <plan-item> [, <plan-item> ...])

<merged-item> ::= 
    [SORT] MERGE (<sorted-item>, <sorted-item> [, <sorted-item> ...])

<hash-item> ::= 
    HASH (<plan-item>, <plan-item> [, <plan-item> ...])

<plan-item> ::= <basic-item> | <plan-expr>

<basic-item> ::= 
    <relation> { NATURAL 
        | INDEX (<indexlist>)
        | ORDER index [INDEX (<indexlist>)] }

<relation> ::= table | view [table]

<indexlist> ::= index [, index ...]
```
Every time a user submits a query to the Firebird engine, the optimizer computes a data retrieval strategy. Most Firebird clients can make this retrieval plan visible to the user. In Firebird’s own isql utility, this is done with the command SET PLAN ON. If you are studying query plans rather than running queries, SET PLANONLY ON will show the plan without executing the query. Use SET PLANONLY OFF to execute the query and show the plan.

A more detailed plan can be obtained when you enable an advanced plan. In isql this can be done with SET EXPLAIN ON. The advanced plan displays more detailed information about the access methods used by the optimizer, however it cannot be included in the PLAN clause of a statement. The description of the advanced plan is beyond the scope of this Language Reference.

In most situations, you can trust that Firebird will select the optimal query plan for you. However, if you have complicated queries that seem to be underperforming, it may very well be worth your while to examine the plan and see if you can improve on it.

**Simple plans**

The simplest plans consist of just a relation name followed by a retrieval method. For example, for an unsorted single-table select without a WHERE clause:

```
select * from students
plan (students natural);
```

Advanced plan:

```
Select Expression
   -> Table "STUDENTS" Full Scan
```

If there’s a WHERE or a HAVING clause, you can specify the index to be used for finding matches:

```
select * from students
   where class = '3C'
plan (students index (ix_stud_class));
```

Advanced plan:

```
Select Expression
```
The INDEX directive is also used for join conditions (to be discussed a little later). It can contain a list of indexes, separated by commas.

ORDER specifies the index for sorting the set if an ORDER BY or GROUP BY clause is present:

```
select * from students
plan (students order pk_students)
order by id;
```

Advanced plan:

```
Select Expression
- Table "STUDENTS" Access By ID
  Index "PK_STUDENTS" Full Scan
```

ORDER and INDEX can be combined:

```
select * from students
  where class >= '3'
  plan (students order pk_students index (ix_stud_class))
  order by id;
```

Advanced plan:

```
Select Expression
  Filter
  Table "STUDENTS" Access By ID
    Index "PK_STUDENTS" Full Scan
  Bitmap
    Index "IX_STUD_CLASS" Range Scan (lower bound: 1/1)
```

It is perfectly OK if ORDER and INDEX specify the same index:

```
select * from students
  where class >= '3'
  plan (students order ix_stud_class index (ix_stud_class))
  order by class;
```

Advanced plan:
For sorting sets when there's no usable index available (or if you want to suppress its use), leave out ORDER and prepend the plan expression with SORT:

```sql
select * from students
plan sort (students natural)
order by name;
```

Advanced plan:

```sql
Select Expression
-> Sort (record length: 128, key length: 56)
  -> Table "STUDENTS" Full Scan
```

Or when an index is used for the search:

```sql
select * from students
where class >= '3'
plan sort (students index (ix_stud_class))
order by name;
```

Advanced plan:

```sql
select * from freshmen
```

Notice that SORT, unlike ORDER, is outside the parentheses. This reflects the fact that the data rows are retrieved unordered and sorted afterwards by the engine.

When selecting from a view, specify the view and the table involved. For instance, if you have a view FRESHMEN that selects just the first-year students:

```sql
select * from freshmen
```
plan (freshmen students natural);

Advanced plan:

Select Expression
  -> Table "STUDENTS" as "FRESHMEN" Full Scan

Or, for instance:

```
select * from freshmen
  where id > 10
plan sort (freshmen students index (pk_students))
  order by name desc;
```

Advanced plan:

Select Expression
  -> Sort (record length: 144, key length: 24)
  -> Filter
    -> Table "STUDENTS" as "FRESHMEN" Access By ID
    -> Bitmap
      -> Index "PK_STUDENTS" Range Scan (lower bound: 1/1)

If a table or view has been aliased, it is the alias, not the original name, that must be used in the PLAN clause.

Composite plans

When a join is made, you can specify the index which is to be used for matching. You must also use the JOIN directive on the two streams in the plan:

```
select s.id, s.name, s.class, c.mentor
  from students s
  join classes c on c.name = s.class
plan join (s natural, c index (pk_classes));
```

Advanced plan:

Select Expression
  -> Nested Loop Join (inner)
  -> Table "STUDENTS" as "S" Full Scan
  -> Filter
    -> Table "CLASSES" as "C" Access By ID
    -> Bitmap
The same join, sorted on an indexed column:

```sql
select s.id, s.name, s.class, c.mentor
from students s
join classes c on c.name = s.class
plan join (s order pk_students, c index (pk_classes))
order by s.id;
```

Advanced plan:

Select Expression
- Nested Loop Join (inner)
  - Table "STUDENTS" as "S" Access By ID
  - Index "PK_STUDENTS" Full Scan
  - Filter
    - Table "CLASSES" as "C" Access By ID
      - Bitmap
        - Index "PK_CLASSES" Unique Scan

And on a non-indexed column:

```sql
select s.id, s.name, s.class, c.mentor
from students s
join classes c on c.name = s.class
plan sort (join (s natural, c index (pk_classes)))
order by s.name;
```

Advanced plan:

Select Expression
- Sort (record length: 152, key length: 12)
  - Nested Loop Join (inner)
    - Table "STUDENTS" as "S" Full Scan
    - Filter
      - Table "CLASSES" as "C" Access By ID
        - Bitmap
          - Index "PK_CLASSES" Unique Scan

With a search condition added:

```sql
select s.id, s.name, s.class, c.mentor
from students s
join classes c on c.name = s.class
```
where s.class <= '2'
plan sort (join (s index (fk_student_class), c index (pk_classes)))
order by s.name;

Advanced plan:

Select Expression
  -> Sort (record length: 152, key length: 12)
  -> Nested Loop Join (inner)
    -> Filter
      -> Table "STUDENTS" as "S" Access By ID
        -> Bitmap
          -> Index "FK_STUDENT_CLASS" Range Scan (lower bound: 1/1)
    -> Filter
      -> Table "CLASSES" as "C" Access By ID
        -> Bitmap
          -> Index "PK_CLASSES" Unique Scan

As a left outer join:

select s.id, s.name, s.class, c.mentor
from classes c
left join students s on c.name = s.class
where s.class <= '2'
plan sort (join (c natural, s index (fk_student_class)))
order by s.name;

Advanced plan:

Select Expression
  -> Sort (record length: 192, key length: 56)
  -> Filter
    -> Nested Loop Join (outer)
      -> Table "CLASSES" as "C" Full Scan
      -> Filter
        -> Table "STUDENTS" as "S" Access By ID
          -> Bitmap
            -> Index "FK_STUDENT_CLASS" Range Scan (full match)

If there are no indices available to match the join condition (or if you don’t want to use it), then it is possible connect the streams using HASH or MERGE method.

To connect using the HASH method in the plan, the HASH directive is used instead of the JOIN directive. In this case, the smaller (secondary) stream is materialized completely into an internal buffer. While reading this secondary stream, a hash function is applied and a pair \( \{hash, pointer to buffer\} \) is written to a hash table. Then the primary stream is read and its hash key is tested against the hash table.
select *
from students s
join classes c on c.cookie = s.cookie
plan hash (c natural, s natural)

Advanced plan:

Select Expression
  -> Filter
     -> Hash Join (inner)
        -> Table "STUDENTS" as "S" Full Scan
        -> Record Buffer (record length: 145)
        -> Table "CLASSES" as "C" Full Scan

For a MERGE join, the plan must first sort both streams on their join column(s) and then merge. This is achieved with the SORT directive (which we've already seen) and MERGE instead of JOIN:

select * from students s
join classes c on c.cookie = s.cookie
plan merge (sort (c natural), sort (s natural));

Adding an ORDER BY clause means the result of the merge must also be sorted:

select * from students s
join classes c on c.cookie = s.cookie
plan sort (merge (sort (c natural), sort (s natural)))
order by c.name, s.id;

Finally, we add a search condition on two indexable columns of table STUDENTS:

select * from students s
join classes c on c.cookie = s.cookie
where s.id < 10 and s.class <= '2'
plan sort (merge (sort (c natural), sort (s index (pk_students, fk_student_class))))
order by c.name, s.id;

As follows from the formal syntax definition, JOINs and MERGEs in the plan may combine more than two streams. Also, every plan expression may be used as a plan item in an encompassing plan. This means that plans of certain complicated queries may have various nesting levels.

Finally, instead of MERGE you may also write SORT MERGE. As this makes absolutely no difference and may create confusion with “real” SORT directives (the ones that do make a difference), it's probably best to stick to plain MERGE.
In addition to the plan for the main query, you can specify a plan for each subquery. For example, the following query with multiple plans will work:

```sql
select *
from color
where exists (
    select *
    from hors
    where horse.code_color = color.code_color
    plan (horse index (fk_horse_color)))
plan (color natural)
```

Occasionally, the optimizer will accept a plan and then not follow it, even though it does not reject it as invalid. One such example was

```
MERGE (unsorted stream, unsorted stream)
```

It is advisable to treat such as plan as “deprecated”.

### 6.1.8. UNION

The `UNION` clause concatenates two or more datasets, thus increasing the number of rows but not the number of columns. Datasets taking part in a `UNION` must have the same number of columns, and columns at corresponding positions must be of the same type. Other than that, they may be totally unrelated.

By default, a union suppresses duplicate rows. `UNION ALL` shows all rows, including any duplicates. The optional `DISTINCT` keyword makes the default behaviour explicit.

**Syntax**

```sql
<union> ::= 
    <individual-select>
    UNION [{DISTINCT | ALL}]
    <individual-select>
    [ 
        [UNION [{DISTINCT | ALL}]
        <individual-select>
        ...
    ]
    [<union-wide-clauses>]

<individual-select> ::= 
    SELECT 
    [TRANSACTION name]
    [FIRST m] [SKIP n]
    [{DISTINCT | ALL}] <columns>
    [INTO <host-varlist>]
```
Unions take their column names from the first select query. If you want to alias union columns, do so in the column list of the topmost SELECT. Aliases in other participating selects are allowed and may even be useful, but will not propagate to the union level.

If a union has an ORDER BY clause, the only allowed sort items are integer literals indicating 1-based column positions, optionally followed by an ASC/DESC and/or a NULLS {FIRST | LAST} directive. This also implies that you cannot order a union by anything that isn’t a column in the union. (You can, however, wrap it in a derived table, which gives you back all the usual sort options.)

Unions are allowed in subqueries of any kind and can themselves contain subqueries. They can also contain joins, and can take part in a join when wrapped in a derived table.

**Examples**

This query presents information from different music collections in one dataset using unions:

```sql
select id, title, artist, length, 'CD' as medium
  from cds
union
select id, title, artist, length, 'LP'
  from records
union
select id, title, artist, length, 'MC'
  from cassettes
order by 3, 2  -- artist, title;
```

If id, title, artist and length are the only fields in the tables involved, the query can also be written as:

```sql
select c.*, 'CD' as medium
  from cds c
union
```
Qualifying the “stars” is necessary here because they are not the only item in the column list. Notice how the “c” aliases in the first and third select do not conflict with each other: their scopes are not union-wide but apply only to their respective select queries.

The next query retrieves names and phone numbers from translators and proofreaders. Translators who also work as proofreaders will show up only once in the result set, provided their phone number is the same in both tables. The same result can be obtained without \texttt{DISTINCT}. With \texttt{ALL}, these people would appear twice.

\begin{verbatim}
select name, phone from translators
union distinct
select name, telephone from proofreaders;
\end{verbatim}

A \texttt{UNION} within a subquery:

\begin{verbatim}
select name, phone, hourly_rate from clowns
where hourly_rate < all
  (select hourly_rate from jugglers
   union
   select hourly_rate from acrobats)
order by hourly_rate;
\end{verbatim}

6.1.9. ORDER BY

When a SELECT statement is executed, the result set is not sorted in any way. It often happens that rows appear to be sorted chronologically, simply because they are returned in the same order they were added to the table by \texttt{INSERT} statements. This is not something you should rely on: the order may change depending on the plan or updates to rows, etc. To specify an explicit sorting order for the set specification, an \texttt{ORDER BY} clause is used.

\textit{Syntax}

\begin{verbatim}
SELECT ... FROM ...
...
ORDER BY \textit{ordering-item} [, \textit{ordering-item} ...]
\end{verbatim}

\begin{verbatim}
<ordering-item> ::= \{col-name | col-alias | col-position | \textit{expression}\}
\end{verbatim}

\begin{verbatim}
[COLLATE \textit{collation-name}]
[ASCENDING | DESCENDING]
\end{verbatim}
Table 75. Arguments for the ORDER BY Clause

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>col-name</td>
<td>Full column name</td>
</tr>
<tr>
<td>col-alias</td>
<td>Column alias</td>
</tr>
<tr>
<td>col-position</td>
<td>Column position in the SELECT list</td>
</tr>
<tr>
<td>expression</td>
<td>Any expression</td>
</tr>
<tr>
<td>collation-name</td>
<td>Collation name (sorting order for string types)</td>
</tr>
</tbody>
</table>

The ORDER BY consists of a comma-separated list of the columns on which the result data set should be sorted. The sort order can be specified by the name of the column — but only if the column was not previously aliased in the SELECT columns list. The alias must be used if it was used in the select list. The ordinal position number of the column in the SELECT column list, the alias given to the column in the SELECT list with the help of the AS keyword, or the number of the column in the SELECT list can be used without restriction.

The three forms of expressing the columns for the sort order can be mixed in the same ORDER BY clause. For instance, one column in the list can be specified by its name and another column can be specified by its number.

- If you sort by column position or alias, then the expression corresponding to this position (alias) will be copied from the SELECT list. This also applies to subqueries, thus, the subquery will be executed at least twice.
- If you use the column position to specify the sort order for a query of the SELECT * style, the server expands the asterisk to the full column list in order to determine the columns for the sort. It is, however, considered “sloppy practice” to design ordered sets this way.

**Sorting Direction**

The keyword ASCENDING — usually abbreviated to ASC — specifies a sort direction from lowest to highest. ASCENDING is the default sort direction.

The keyword DESCENDING — usually abbreviated to DESC — specifies a sort direction from highest to lowest.

Specifying ascending order for one column and descending order for another is allowed.

**Collation Order**

The keyword COLLATE specifies the collation order for a string column if you need a collation that is different from the normal one for this column. The normal collation order will be either the default one for the database character set, or the one set explicitly in the column's definition.
The keyword NULLS defines where NULL in the associated column will fall in the sort order: NULLS FIRST places the rows with the NULL column above rows ordered by that column’s value; NULLS LAST places those rows after the ordered rows.

NULLS FIRST is the default.

**Ordering UNION-ed Sets**

The discrete queries contributing to a UNION cannot take an ORDER BY clause. The only option is to order the entire output, using one ORDER BY clause at the end of the overall query.

The simplest — and, in some cases, the only — method for specifying the sort order is by the ordinal column position. However, it is also valid to use the column names or aliases, from the first contributing query only.

The ASC/DESC and/or NULLS directives are available for this global set.

If discrete ordering within the contributing set is required, use of derived tables or common table expressions for those sets may be a solution.

**Examples of ORDER BY**

Sorting the result set in ascending order, ordering by the RDB$CHARACTER_SET_ID and RDB$COLLATION_ID columns of the RDB$COLLATIONS table:

```sql
SELECT
    RDB$CHARACTER_SET_ID AS CHARSET_ID,
    RDB$COLLATION_ID AS COLL_ID,
    RDB$COLLATION_NAME AS NAME
FROM RDB$COLLATIONS
ORDER BY RDB$CHARACTER_SET_ID, RDB$COLLATION_ID;
```

The same, but sorting by the column aliases:

```sql
SELECT
    RDB$CHARACTER_SET_ID AS CHARSET_ID,
    RDB$COLLATION_ID AS COLL_ID,
    RDB$COLLATION_NAME AS NAME
FROM RDB$COLLATIONS
ORDER BY CHARSET_ID, COLL_ID;
```

Sorting the output data by the column position numbers:

```sql
SELECT
    RDB$CHARACTER_SET_ID AS CHARSET_ID,
    RDB$COLLATION_ID AS COLL_ID,
```
Sorting a `SELECT *` query by position numbers — possible, but *nasty* and not recommended:

```sql
SELECT *
FROM RDB$COLLATIONS
ORDER BY 3, 2;
```

Sorting by the second column in the `BOOKS` table, or—if `BOOKS` has only one column—the `FILMS.DIRECTOR` column:

```sql
SELECT
    BOOKS.*,
    FILMS.DIRECTOR
FROM BOOKS, FILMS
ORDER BY 2;
```

Sorting in descending order by the values of column `PROCESS_TIME`, with `NULLs` placed at the beginning of the set:

```sql
SELECT *
FROM MSG
ORDER BY PROCESS_TIME DESC NULLS FIRST;
```

Sorting the set obtained by a `UNION` of two queries. Results are sorted in descending order for the values in the second column, with `NULLs` at the end of the set; and in ascending order for the values of the first column with `NULLs` at the beginning.

```sql
SELECT
    DOC_NUMBER, DOC_DATE
FROM PAYORDER
UNION ALL
SELECT
    DOC_NUMBER, DOC_DATE
FROM BUDGORDER
ORDER BY 2 DESC NULLS LAST, 1 ASC NULLS FIRST;
```

### 6.1.10. ROWS

*Used for*

Retrieving a slice of rows from an ordered set

*Available in*
Table 76. Arguments for the ROWS Clause

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m, n</td>
<td>Any integer expressions</td>
</tr>
</tbody>
</table>

**ROWS is non-standard syntax**

ROWS is a Firebird-specific clause. Use the SQL-standard OFFSET, FETCH syntax wherever possible.

Limits the amount of rows returned by the SELECT statement to a specified number or range.

The ROWS clause also does the same job as the FIRST and SKIP clauses, but neither are SQL-compliant. Unlike FIRST and SKIP, and OFFSET and FETCH, the ROWS and TO clauses accept any type of integer expression as their arguments, without parentheses. Of course, parentheses may still be needed for nested evaluations inside the expression, and a subquery must always be enclosed in parentheses.

- Numbering of rows in the intermediate set—the overall set cached on disk before the “slice” is extracted—starts at 1.
- OFFSET/FETCH, FIRST/Skip, and ROWS can all be used without the ORDER BY clause, although it rarely makes sense to do so—except perhaps when you want to take a quick look at the table data and don't care that rows will be in a non-deterministic order. For this purpose, a query like “SELECT * FROM TABLE1 ROWS 20” would return the first 20 rows instead of a whole table that might be rather big.

Calling ROWS \( m \) retrieves the first \( m \) records from the set specified.

**Characteristics of using ROWS \( m \) without a TO clause:**

- If \( m \) is greater than the total number of records in the intermediate data set, the entire set is returned
- If \( m = 0 \), an empty set is returned
- If \( m < 0 \), the SELECT statement call fails with an error

Calling ROWS \( m \) TO \( n \) retrieves the rows from the set, starting at row \( m \) and ending after row \( n \)—the set is inclusive.
Characteristics of using `ROWS m` with a `TO` clause:

- If \( m \) is greater than the total number of rows in the intermediate set and \( n \geq m \), an empty set is returned.
- If \( m \) is not greater than \( n \) and \( n \) is greater than the total number of rows in the intermediate set, the result set will be limited to rows starting from \( m \), up to the end of the set.
- If \( m < 1 \) and \( n < 1 \), the SELECT statement call fails with an error.
- If \( n = m - 1 \), an empty set is returned.
- If \( n < m - 1 \), the SELECT statement call fails with an error.

Using a `TO` clause without a `ROWS` clause:

While `ROWS` replaces the `FIRST` and `SKIP` syntax, there is one situation where the `ROWS` syntax does not provide the same behaviour: specifying `SKIP n` on its own returns the entire intermediate set, without the first \( n \) rows. The `ROWS ... TO` syntax needs a little help to achieve this.

With the `ROWS` syntax, you need a `ROWS` clause in association with the `TO` clause and deliberately make the second \((n)\) argument greater than the size of the intermediate data set. This is achieved by creating an expression for \( n \) that uses a subquery to retrieve the count of rows in the intermediate set and adds 1 to it.

Replacing of `FIRST/SKIP` and `OFFSET/FETCH`

The `ROWS` clause can be used instead of the SQL-standard `OFFSET/FETCH` or non-standard `FIRST/SKIP` clauses, except the case where only `OFFSET` or `SKIP` is used, that is when the whole result set is returned except for skipping the specified number of rows from the beginning.

In order to implement this behaviour using `ROWS`, you must specify the `TO` clause with a value larger than the size of the returned result set.

Mixing `ROWS` and `FIRST/SKIP` or `OFFSET/FETCH`

`ROWS` syntax cannot be mixed with `FIRST/SKIP` or `OFFSET/FETCH` in the same SELECT expression. Using the different syntaxes in different subqueries in the same statement is allowed.

`ROWS` Syntax in `UNION` Queries

When `ROWS` is used in a `UNION` query, the `ROWS` directive is applied to the unioned set and must be placed after the last `SELECT` statement.

If a need arises to limit the subsets returned by one or more `SELECT` statements inside `UNION`, there are a couple of options:

1. Use `FIRST/SKIP` syntax in these `SELECT` statements — bearing in mind that an ordering clause (`ORDER BY`) cannot be applied locally to the discrete queries, but only to the combined output.
2. Convert the queries to derived tables with their own `ROWS` clauses.
Examples of \texttt{ROWS}

The following examples rewrite the \texttt{examples} used in the section about \texttt{FIRST} and \texttt{SKIP}, \textit{earlier in this chapter}.

Retrieve the first ten names from the output of a sorted query on the \texttt{PEOPLE} table:

\begin{verbatim}
SELECT id, name
FROM People
ORDER BY name ASC
ROWS 1 TO 10;
\end{verbatim}

or its equivalent

\begin{verbatim}
SELECT id, name
FROM People
ORDER BY name ASC
ROWS 10;
\end{verbatim}

Return all records from the \texttt{PEOPLE} table except for the first 10 names:

\begin{verbatim}
SELECT id, name
FROM People
ORDER BY name ASC
ROWS 11 TO (SELECT COUNT(*) FROM People);
\end{verbatim}

And this query will return the last 10 records (pay attention to the parentheses):

\begin{verbatim}
SELECT id, name
FROM People
ORDER BY name ASC
ROWS (SELECT COUNT(*) - 9 FROM People)
TO (SELECT COUNT(*) FROM People);
\end{verbatim}

This one will return rows 81-100 from the \texttt{PEOPLE} table:

\begin{verbatim}
SELECT id, name
FROM People
ORDER BY name ASC
ROWS 81 TO 100;
\end{verbatim}

\textbf{ROWS} can also be used with the \texttt{UPDATE} and \texttt{DELETE} statements.

\textit{See also} \\
\texttt{FIRST, SKIP, OFFSET, FETCH}
6.1.11. OFFSET, FETCH

Used for
Retrieving a slice of rows from an ordered set

Available in
DSQL, PSQL

Syntax

```
SELECT <columns> FROM ... 
[WHERE ...] 
[ORDER BY ...] 
[OFFSET <m> {ROW | ROWS}] 
[FETCH {FIRST | NEXT} [ <n> ] { ROW | ROWS } ONLY]
```

```
<m>, <n> ::= 
<integer-literal> 
| <query-parameter>
```

Table 77. Arguments for the OFFSET and FETCH Clause

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer-literal</td>
<td>Integer literal</td>
</tr>
<tr>
<td>query-parameter</td>
<td>Query parameter place-holder. ? in DSQL and :paramname in PSQL</td>
</tr>
</tbody>
</table>

The OFFSET and FETCH clauses are an SQL:2008 compliant equivalent for FIRST/Skip, and an alternative for ROWS. The OFFSET clause specifies the number of rows to skip. The FETCH clause specifies the number of rows to fetch.

When <n> is left out of the FETCH clause (eg FETCH FIRST ROW ONLY), one row will be fetched.

The choice between ROW or ROWS, or FIRST or NEXT in the clauses is just for aesthetic purposes (eg making the query more readable or grammatically correct). Technically there is no difference between OFFSET 10 ROW or OFFSET 10 ROWS, or FETCH NEXT 10 ROWS ONLY or FETCH FIRST 10 ROWS ONLY.

As with SKIP and FIRST, OFFSET and FETCH clauses can be applied independently, in both top-level and nested query expressions.

1. Firebird doesn’t support the percentage FETCH defined in the SQL standard.
2. Firebird doesn’t support the FETCH … WITH TIES defined in the SQL standard.
3. The FIRST/Skip and ROWS clause are non-standard alternatives.
4. The OFFSET and/or FETCH clauses cannot be combined with ROWS or FIRST/Skip on the same query expression.
5. Expressions, column references, etc are not allowed within either clause.
6. Contrary to the ROWS clause, OFFSET and FETCH are only available on SELECT statements.
Examples of OFFSET and FETCH

Return all rows except the first 10, ordered by column COL1

```
SELECT *
FROM T1
ORDER BY COL1
OFFSET 10 ROWS
```

Return the first 10 rows, ordered by column COL1

```
SELECT *
FROM T1
ORDER BY COL1
FETCH FIRST 10 ROWS ONLY
```

Using OFFSET and FETCH clauses in a derived table and in the outer query

```
SELECT *
FROM (  
    SELECT *
    FROM T1
    ORDER BY COL1 DESC
    OFFSET 1 ROW
    FETCH NEXT 10 ROWS ONLY
    ) a
ORDER BY a.COL1
FETCH FIRST ROW ONLY
```

The following examples rewrite the FIRST/Skip examples and ROWS examples earlier in this chapter.

Retrieve the first ten names from the output of a sorted query on the PEOPLE table:

```
SELECT id, name
FROM People
ORDER BY name ASC
FETCH NEXT 10 ROWS ONLY;
```

Return all records from the PEOPLE table except for the first 10 names:

```
SELECT id, name
FROM People
ORDER BY name ASC
OFFSET 10 ROWS;
```

And this query will return the last 10 records. Contrary to FIRST/SKIP and ROWS we cannot use expressions (including sub-queries). To retrieve the last 10 rows, reverse the sort to the first (last)
10 rows, and then sort in the right order.

```
SELECT id, name
FROM (  
    SELECT id, name
    FROM People
    ORDER BY name DESC
    FETCH FIRST 10 ROWS ONLY
  ) a
ORDER BY name ASC;
```

This one will return rows 81-100 from the PEOPLE table:

```
SELECT id, name
FROM People
ORDER BY name ASC
OFFSET 80 ROWS
FETCH NEXT 20 ROWS;
```

See also
FIRST, SKIP, ROWS

6.1.12. FOR UPDATE [OF]

Syntax

```
SELECT ... FROM single_table  
[WHERE ...]  
[FOR UPDATE [OF <column_list>]]
```

FOR UPDATE does not do what its name suggests. It’s only effect currently is to disable the pre-fetch buffer.

It is likely to change in future: the plan is to validate cursors marked with FOR UPDATE if they are truly updateable and reject positioned updates and deletes for cursors evaluated as non-updateable.

The 0f sub-clause does not do anything at all.

6.1.13. WITH LOCK

Used for
Limited pessimistic locking

Available in
DSQL, PSQL
WITH LOCK provides a limited explicit pessimistic locking capability for cautious use in conditions where the affected row set is:

a. extremely small (ideally singleton), and

b. precisely controlled by the application code.

This is for experts only!

The need for a pessimistic lock in Firebird is very rare indeed and should be well understood before use of this extension is considered.

It is essential to understand the effects of transaction isolation and other transaction attributes before attempting to implement explicit locking in your application.

If the WITH LOCK clause succeeds, it will secure a lock on the selected rows and prevent any other transaction from obtaining write access to any of those rows, or their dependants, until your transaction ends.

WITH LOCK can only be used with a top-level, single-table SELECT statement. It is not available:

- in a subquery specification
- for joined sets
- with the DISTINCT operator, a GROUP BY clause or any other aggregating operation
- with a view
- with the output of a selectable stored procedure
- with an external table
- with a UNION query

As the engine considers, in turn, each record falling under an explicit lock statement, it returns either the record version that is the most currently committed, regardless of database state when the statement was submitted, or an exception.

Wait behaviour and conflict reporting depend on the transaction parameters specified in the TPB block:

Table 78. How TPB settings affect explicit locking
### TPB mode

<table>
<thead>
<tr>
<th>TPB mode</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>isc_tpb_consistency</td>
<td>Explicit locks are overridden by implicit or explicit table-level locks and are ignored.</td>
</tr>
<tr>
<td>isc_tpb_concurrency + isc_tpb_nowait</td>
<td>If a record is modified by any transaction that was committed since the transaction attempting to get explicit lock started, or an active transaction has performed a modification of this record, an update conflict exception is raised immediately.</td>
</tr>
<tr>
<td>isc_tpb_concurrency + isc_tpb_wait</td>
<td>If the record is modified by any transaction that has committed since the transaction attempting to get explicit lock started, an update conflict exception is raised immediately.</td>
</tr>
<tr>
<td>isc_tpb_read_committed + isc_tpb_nowait</td>
<td>If there is an active transaction holding ownership on this record (via explicit locking or normal update), an update conflict exception is raised immediately.</td>
</tr>
<tr>
<td>isc_tpb_read_committed + isc_tpb_wait</td>
<td>If there is an active transaction holding ownership on this record (via explicit locking or normal optimistic write-lock), the transaction attempting the explicit lock waits for the outcome of blocking transaction and, when it finishes, attempts to get the lock on the record again. Update conflict exceptions can never be raised by an explicit lock statement in this TPB mode.</td>
</tr>
</tbody>
</table>

#### Usage with a `FOR UPDATE` Clause

If the `FOR UPDATE` sub-clause precedes the `WITH LOCK` sub-clause, buffered fetches are suppressed. Thus, the lock will be applied to each row, one by one, at the moment it is fetched. It becomes possible, then, that a lock which appeared to succeed when requested will nevertheless fail subsequently, when an attempt is made to fetch a row which has become locked by another transaction in the meantime.

As an alternative, it may be possible in your access components to set the size of the fetch buffer to 1. This would enable you to process the currently-locked row before the next is fetched and locked, or to handle errors without rolling back your transaction.

**OF `<column_list>`**

This optional sub-clause does nothing at all.

See also
FOR UPDATE [OF]

How the engine deals with WITH LOCK

When an UPDATE statement tries to access a record that is locked by another transaction, it either raises an update conflict exception or waits for the locking transaction to finish, depending on TPB mode. Engine behaviour here is the same as if this record had already been modified by the locking transaction.

No special gdscodes are returned from conflicts involving pessimistic locks.

The engine guarantees that all records returned by an explicit lock statement are actually locked and do meet the search conditions specified in WHERE clause, as long as the search conditions do not depend on any other tables, via joins, subqueries, etc. It also guarantees that rows not meeting the search conditions will not be locked by the statement. It can not guarantee that there are no rows which, though meeting the search conditions, are not locked.

This situation can arise if other, parallel transactions commit their changes during the course of the locking statement's execution.

The engine locks rows at fetch time. This has important consequences if you lock several rows at once. Many access methods for Firebird databases default to fetching output in packets of a few hundred rows (“buffered fetches”). Most data access components cannot bring you the rows contained in the last-fetched packet, when an error occurred.

Caveats using WITH LOCK

• Rolling back of an implicit or explicit savepoint releases record locks that were taken under that savepoint, but it doesn’t notify waiting transactions. Applications should not depend on this behaviour as it may get changed in the future.

• While explicit locks can be used to prevent and/or handle unusual update conflict errors, the volume of deadlock errors will grow unless you design your locking strategy carefully and control it rigorously.

• Most applications do not need explicit locks at all. The main purposes of explicit locks are:

  1. to prevent expensive handling of update conflict errors in heavily loaded applications, and
  2. to maintain integrity of objects mapped to a relational database in a clustered environment.

If your use of explicit locking doesn’t fall in one of these two categories, then it’s the wrong way to do the task in Firebird.

• Explicit locking is an advanced feature; do not misuse it! While solutions for these kinds of problems may be very important for web sites handling thousands of concurrent writers, or for ERP/CRM systems operating in large corporations, most application programs do not need to work in such conditions.

Examples using explicit locking

i. Simple:
ii. Multiple rows, one-by-one processing with DSQL cursor:

```
SELECT * FROM DOCUMENT WHERE PARENT_ID=?
FOR UPDATE WITH LOCK;
```

### 6.1.14. INTO

**Used for**

Passing SELECT output into variables

**Available in**

PSQL

**Syntax**

In PSQL the INTO clause is placed at the very end of the SELECT statement.

```
SELECT [...] <column-list>
FROM ...
[...]
[INTO <variable-list>]

<variable-list> ::= [:]psqlvar [, [:]psqlvar ...]
```

The colon prefix before local variable names in PSQL is optional in the INTO clause.

In PSQL code (triggers, stored procedures and executable blocks), the results of a SELECT statement can be loaded row-by-row into local variables. It is often the only way to do anything with the returned values at all, unless an explicit or implicit cursor name is specified. The number, order and types of the variables must match the columns in the output row.

A “plain” SELECT statement can only be used in PSQL if it returns at most one row, i.e., if it is a singleton select. For multi-row selects, PSQL provides the FOR SELECT loop construct, discussed later in the PSQL chapter. PSQL also supports the DECLARE CURSOR statement, which binds a named cursor to a SELECT statement. The cursor can then be used to walk the result set.

**Examples**

1. Selecting some aggregated values and passing them into previously declared variables `min_amt`, `avg_amt` and `max_amt`:

```
SELECT min(amount), avg(cast(amount as float)), max(amount)
FROM orders
WHERE artno = 372218
```
The CAST serves to make the average a real number; otherwise, since amount is presumably an integer field, SQL rules would truncate it to the nearest lower integer.

2. A PSQL trigger that retrieves two values as a BLOB field (using the LIST() function) and assigns it INTO a third field:

```sql
select list(name, ',', '')
from persons p
where p.id in (new.father, new.mother)
into new.parentnames;
```

### 6.1.15. Common Table Expressions (“WITH … AS … SELECT”)

**Available in**
DSQL, PSQL

**Syntax**

```
<cte-construct> ::= 
<cte-defs> 
<main-query>

<cte-defs> ::= WITH [RECURSIVE] <cte> [, <cte> ...]

<cte> ::= name [([<column-list>]] AS (<cte-stmt>)

<column-list> ::= column-alias [, column-alias ...]
```

**Table 79. Arguments for Common Table Expressions**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cte-stmt</td>
<td>Any SELECT statement, including UNION</td>
</tr>
<tr>
<td>main-query</td>
<td>The main SELECT statement, which can refer to the CTEs defined in the preamble</td>
</tr>
<tr>
<td>name</td>
<td>Alias for a table expression</td>
</tr>
<tr>
<td>column-alias</td>
<td>Alias for a column in a table expression</td>
</tr>
</tbody>
</table>

A common table expression or CTE can be described as a virtual table or view, defined in a preamble to a main query, and going out of scope after the main query's execution. The main query can reference any CTEs defined in the preamble as if they were regular tables or views. CTEs can be recursive, i.e. self-referencing, but they cannot be nested.
CTE Notes

- A CTE definition can contain any legal SELECT statement, as long as it doesn’t have a “WITH…” preamble of its own (no nesting).

- CTEs defined for the same main query can reference each other, but care should be taken to avoid loops.

- CTEs can be referenced from anywhere in the main query.

- Each CTE can be referenced multiple times in the main query, using different aliases if necessary.

- When enclosed in parentheses, CTE constructs can be used as subqueries in SELECT statements, but also in UPDATES, MERGEs etc.

- In PSQL, CTEs are also supported in FOR loop headers:

```sql
for
  with my_rivers as (select * from rivers where owner = 'me')
  select name, length from my_rivers into :rname, :rlen
do
begin
...
end
```

In Firebird 3.0.2 and earlier, if a CTE is declared, it must be used later: otherwise, you will get an error like this: “CTE "AAA" is not used in query”.

This restriction was removed in Firebird 3.0.3.

Example

```sql
with dept_year_budget as (  
  select fiscal_year,  
    dept_no,  
    sum(projected_budget) as budget  
  from proj_dept_budget  
  group by fiscal_year, dept_no  
)
select d.dept_no,  
  d.department,  
  dyb_2008.budget as budget_08,  
  dyb_2009.budget as budget_09  
from department d  
left join dept_year_budget dyb_2008  
on d.dept_no = dyb_2008.dept_no  
  and dyb_2008.fiscal_year = 2008  
left join dept_year_budget dyb_2009  
on d.dept_no = dyb_2009.dept_no  
  and dyb_2009.fiscal_year = 2009  
where exists (  
```
Recursive CTEs

A recursive (self-referencing) CTE is a UNION which must have at least one non-recursive member, called the anchor. The non-recursive member(s) must be placed before the recursive member(s). Recursive members are linked to each other and to their non-recursive neighbour by UNION ALL operators. The unions between non-recursive members may be of any type.

Recursive CTEs require the RECURSIVE keyword to be present right after WITH. Each recursive union member may reference itself only once, and it must do so in a FROM clause.

A great benefit of recursive CTEs is that they use far less memory and CPU cycles than an equivalent recursive stored procedure.

Execution Pattern

The execution pattern of a recursive CTE is as follows:

- The engine begins execution from a non-recursive member.
- For each row evaluated, it starts executing each recursive member one by one, using the current values from the outer row as parameters.
- If the currently executing instance of a recursive member produces no rows, execution loops back one level and gets the next row from the outer result set.

Example of recursive CTEs

```sql
WITH RECURSIVE DEPT_YEAR_BUDGET AS (  
  SELECT  
    FISCAL_YEAR,  
    DEPT_NO,  
    SUM(PROJECTED_BUDGET) BUDGET  
  FROM PROJ_DEPT_BUDGET  
  GROUP BY FISCAL_YEAR, DEPT_NO 
),  
DEPT_TREE AS (  
  SELECT  
    DEPT_NO,  
    HEAD_DEPT,  
    DEPARTMENT,  
    CAST('' AS VARCHAR(255)) AS INDENT  
  FROM DEPARTMENT  
  WHERE HEAD_DEPT IS NULL  
  UNION ALL  
  SELECT  
    D.DEPT_NO,  
    D.HEAD_DEPT,  
```
The next example returns the pedigree of a horse. The main difference is that recursion occurs simultaneously in two branches of the pedigree.

```sql
WITH RECURSIVE PEDIGREE (CODE_HORSE, CODE_FATHER, CODE_MOTHER, NAME, MARK, DEPTH) AS (SELECT HORSE.CODE_HORSE, HORSE.CODE_FATHER, HORSE.CODE_MOTHER, HORSE.NAME, CAST('' AS VARCHAR(80)), 0 FROM HORSE WHERE HORSE.CODE_HORSE = :CODE_HORSE UNION ALL SELECT HORSE.CODE_HORSE, HORSE.CODE_FATHER, HORSE.CODE_MOTHER, HORSE.NAME, 'F' || PEDIGREE.MARK, PEDIGREE.DEPTH + 1 FROM HORSE JOIN PEDIGREE
```
ON HORSE.CODE_HORSE = PEDIGREE.CODE_FATHER
WHERE
  PEDIGREE.DEPTH < :MAX_DEPTH
UNION ALL
SELECT
  HORSE.CODE_HORSE,
  HORSE.CODE_FATHER,
  HORSE.CODE_MOTHER,
  HORSE.NAME,
  'M' || PEDIGREE.MARK,
  PEDIGREE.DEPTH + 1
FROM
  HORSE
JOIN PEDIGREE
  ON HORSE.CODE_HORSE = PEDIGREE.CODE_MOTHER
WHERE
  PEDIGREE.DEPTH < :MAX_DEPTH
)
SELECT
  CODE_HORSE,
  NAME,
  MARK,
  DEPTH
FROM
  PEDIGREE

Notes on recursive CTEs
• Aggregates (DISTINCT, GROUP BY, HAVING) and aggregate functions (SUM, COUNT, MAX etc) are not
  allowed in recursive union members.
• A recursive reference cannot participate in an outer join.
• The maximum recursion depth is 1024.

6.2. INSERT

Used for
Inserting rows of data into a table

Available in
DSQL, ESQL, PSQL

Syntax

INSERT INTO target
  {DEFAULT VALUES | [[<column_list>]] <value_source>}
  [RETURNING <returning_list> [INTO <variables>]]

<column_list> ::= colname [, colname ...]
The **INSERT** statement is used to add rows to a table or to one or more tables underlying a view:

- If the column values are supplied in a **VALUES** clause, exactly one row is inserted.
- The values may be provided instead by a **SELECT** expression, in which case zero to many rows may be inserted.
- With the **DEFAULT VALUES** clause, no values are provided at all and exactly one row is inserted.

### Restrictions

- Columns returned to the **NEW.column_name** context variables in triggers should not have a colon (":") prefixed to their names.
- No column may appear more than once in the column list.

### ALERT: BEFORE INSERT Triggers

Regardless of the method used for inserting rows, be mindful of any columns in the target table or view that are populated by BEFORE INSERT triggers, such as primary keys and case-insensitive search columns. Those columns should be excluded from both the **column_list** and the **VALUES** list if, as they should, the triggers test the **NEW.column_name** for NULL.

### 6.2.1. INSERT ... VALUES

The **VALUES** list must provide a value for every column in the column list, in the same order and of the correct type. The column list need not specify every column in the target but, if the column list...
is absent, the engine requires a value for every column in the table or view (computed columns excluded).

**Introducer syntax** provides a way to identify the character set of a value that is a string constant (literal). Introducer syntax works only with literal strings: it cannot be applied to string variables, parameters, column references or values that are expressions.

**Examples**

```
INSERT INTO cars (make, model, year)
VALUES ('Ford', 'T', 1908);

INSERT INTO cars
VALUES ('Ford', 'T', 1908, 'USA', 850);

-- notice the '_' prefix (introducer syntax)
INSERT INTO People
VALUES (_ISO8859_1 'Hans-Jörg Schäfer');
```

### 6.2.2. **INSERT … SELECT**

For this method of inserting, the output columns of the SELECT statement must provide a value for every target column in the column list, in the same order and of the correct type.

Literal values, context variables or expressions of compatible type can be substituted for any column in the source row. In this case, a source column list and a corresponding VALUES list are required.

If the column list is absent—as it is when SELECT * is used for the source expression—the column_list must contain the names of every column in the target table or view (computed columns excluded).

**Examples**

```
INSERT INTO cars (make, model, year)
SELECT make, model, year
FROM new_cars;

INSERT INTO cars
SELECT * FROM new_cars;

INSERT INTO Members (number, name)
SELECT number, name FROM NewMembers
WHERE Accepted = 1
UNION ALL
SELECT number, name FROM SuspendedMembers
WHERE Vindicated = 1

INSERT INTO numbers(num)
```
WITH RECURSIVE r(n) as 
  SELECT 1 FROM rdb$database 
  UNION ALL 
  SELECT n+1 FROM r WHERE n < 100 
) 
SELECT n FROM r

Of course, the column names in the source table need not be the same as those in the target table. Any type of SELECT statement is permitted, as long as its output columns exactly match the insert columns in number, order and type. Types need not be exactly the same, but they must be assignment-compatible.

When using and INSERT ... SELECT with a RETURNING clause, the SELECT has to produce at most one row, as RETURNING currently only works for statements affecting at most one row.

This behaviour may change in future Firebird versions.

6.2.3. INSERT ... DEFAULT VALUES

The DEFAULT VALUES clause allows insertion of a record without providing any values at all, either directly or from a SELECT statement. This is only possible if every NOT NULL or CHECKed column in the table either has a valid default declared or gets such a value from a BEFORE INSERT trigger. Furthermore, triggers providing required field values must not depend on the presence of input values.

Example

```
INSERT INTO journal 
DEFAULT VALUES 
RETURNING entry_id;
```

6.2.4. The RETURNING clause

An INSERT statement adding at most one row may optionally include a RETURNING clause in order to return values from the inserted row. The clause, if present, need not contain all of the insert columns and may also contain other columns or expressions. The returned values reflect any changes that may have been made in BEFORE INSERT triggers.

The optional INTO sub-clause is only valid in PSQL.

Multiple INSERTs

In DSQL, a statement with RETURNING always returns only one row. If the RETURNING clause is specified and more than one row is inserted by the INSERT statement, the statement fails and an error message is returned. This behaviour may change in future Firebird versions.
Examples

```sql
INSERT INTO Scholars (
    firstname,
    lastname,
    address,
    phone,
    email)
VALUES (
    'Henry',
    'Higgins',
    '27A Wimpole Street',
    '3231212',
    NULL)
RETURNING lastname, fullname, id;

INSERT INTO Dumbbells (firstname, lastname, iq)
    SELECT fname, lname, iq
    FROM Friends
    ORDER BY iq ROWS 1
    RETURNING id, firstname, iq
    INTO :id, :fname, :iq;
```

Notes

- RETURNING is supported for VALUES and DEFAULT VALUES inserts, and singleton SELECT inserts.
- In DSQL, a statement with a RETURNING clause always returns exactly one row. If no record was actually inserted, the fields in this row are all NULL. This behaviour may change in a later version of Firebird. In PSQL, if no row was inserted, nothing is returned, and the target variables keep their existing values.

6.2.5. Inserting into BLOB columns

Inserting into BLOB columns is only possible under the following circumstances:

1. The client application has made special provisions for such inserts, using the Firebird API. In this case, the modus operandi is application-specific and outside the scope of this manual.

2. The value inserted is a string literal of no more than 65,533 bytes (64KB - 3).

A limit, in characters, is calculated at run-time for strings that are in multi-byte character sets, to avoid overrunning the bytes limit. For example, for a UTF8 string (max. 4 bytes/character), the run-time limit is likely to be about (floor(65533/4)) = 16383 characters.

3. You are using the “INSERT … SELECT” form and one or more columns in the result set are BLOBs.
6.3. UPDATE

Used for
Modifying rows in tables and views

Available in
DSQL, ESQL, PSQL

Syntax

```sql
UPDATE target [[AS] alias]
    SET col = <value> [, col = <value> ...]
    [WHERE {<search-conditions> | CURRENT OF cursorname}]
    [PLAN <plan_items>]
    [ORDER BY <sort_items>]
    [ROWS m [TO n]]
    [RETURNING <returning_list> [INTO <variables>]]
```

```sql
```

```sql
<return_value> ::=colname
    | table_or_alias.colname
    | NEW.colname
    | OLD.colname
    | <value>
```

```sql
<variables> ::= [:]varname [, [:]varname ...]
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>The name of the table or view where the records are updated</td>
</tr>
<tr>
<td>alias</td>
<td>Alias for the table or view</td>
</tr>
<tr>
<td>col</td>
<td>Name or alias of a column in the table or view</td>
</tr>
<tr>
<td>value</td>
<td>Expression for the new value for a column that is to be updated in the</td>
</tr>
<tr>
<td></td>
<td>table or view by the statement, or a value to be returned</td>
</tr>
<tr>
<td>search-conditions</td>
<td>A search condition limiting the set of the rows to be updated</td>
</tr>
<tr>
<td>cursorname</td>
<td>The name of the cursor through which the row(s) to be updated are</td>
</tr>
<tr>
<td></td>
<td>positioned</td>
</tr>
<tr>
<td>plan_items</td>
<td>Clauses in the query plan</td>
</tr>
<tr>
<td>sort_items</td>
<td>Columns listed in an ORDER BY clause</td>
</tr>
<tr>
<td>m, n</td>
<td>Integer expressions for limiting the number of rows to be updated</td>
</tr>
<tr>
<td>ret_value</td>
<td>A value to be returned in the RETURNING clause</td>
</tr>
</tbody>
</table>

Table 81. Arguments for the UPDATE Statement Parameters
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varname</td>
<td>Name of a PSQL local variable</td>
</tr>
</tbody>
</table>

The `UPDATE` statement changes values in a table or in one or more of the tables that underlie a view. The columns affected are specified in the `SET` clause. The rows affected may be limited by the `WHERE` and `ROWS` clauses. If neither `WHERE` nor `ROWS` is present, all the records in the table will be updated.

### 6.3.1. Using an alias

If you assign an alias to a table or a view, the alias must be used when specifying columns and also in any column references included in other clauses.

**Example**

Correct usage:

```
update Fruit set soort = 'pisang' where ...
update Fruit set Fruit.soort = 'pisang' where ...
update Fruit F set soort = 'pisang' where ...
update Fruit F set F.soort = 'pisang' where ...
```

Not possible:

```
update Fruit F set Fruit.soort = 'pisang' where ...
```

### 6.3.2. The SET Clause

In the `SET` clause, the assignment phrases, containing the columns with the values to be set, are separated by commas. In an assignment phrase, column names are on the left and the values or expressions containing the assignment values are on the right. A column may be included only once in the `SET` clause.

A column name can be used in expressions on the right. The old value of the column will always be used in these right-side values, even if the column was already assigned a new value earlier in the `SET` clause.

*Here is an example*

Data in the TSET table:

```
A B
---
1 0
2 0
```
The statement:

```
UPDATE tset SET a = 5, b = a;
```

will change the values to:

```
A B
---
5 1
5 2
```

Notice that the old values (1 and 2) are used to update the b column even after the column was assigned a new value (5).

Before Firebird 2.5, columns got their new values immediately upon assignment. It was non-standard behaviour that was fixed in Firebird 2.5.

### 6.3.3. The `WHERE` Clause

The `WHERE` clause sets the conditions that limit the set of records for a searched update.

In PSQL, if a named cursor is being used for updating a set, using the `WHERE CURRENT OF` clause, the action is limited to the row where the cursor is currently positioned. This is a positioned update.

The `WHERE CURRENT OF` clause is available only in PSQL, since there is no statement for creating and manipulating an explicit cursor in DSQL. Searched updates are also available in PSQL, of course.

#### Examples

```
UPDATE People
    SET firstname = 'Boris'
    WHERE lastname = 'Johnson';

UPDATE employee e
    SET salary = salary * 1.05
    WHERE EXISTS(
        SELECT *
        FROM employee_project ep
        WHERE e.emp_no = ep.emp_no);

UPDATE addresses
    SET city = 'Saint Petersburg', citycode = 'PET'
    WHERE city = 'Leningrad'

UPDATE employees
    SET salary = 2.5 * salary
```
WHERE title = 'CEO'

For string literals with which the parser needs help to interpret the character set of the data, the introducer syntax may be used. The string literal is preceded by the character set name, prefixed with an underscore character:

```sql
-- notice the '_' prefix
UPDATE People
SET name = _ISO8859_1 'Hans-Jörg Schäfer'
WHERE id = 53662;
```

### 6.3.4. The ORDER BY and ROWS Clauses

The ORDER BY and ROWS clauses make sense only when used together. However, they can be used separately.

If ROWS has one argument, $m$, the rows to be updated will be limited to the first $m$ rows.

**Points to note**
- If $m >$ the number of rows being processed, the entire set of rows is updated
- If $m = 0$, no rows are updated
- If $m < 0$, an error occurs and the update fails

If two arguments are used, $m$ and $n$, ROWS limits the rows being updated to rows from $m$ to $n$ inclusively. Both arguments are integers and start from 1.

**Points to note**
- If $m >$ the number of rows being processed, no rows are updated
- If $n >$ the number of rows, rows from $m$ to the end of the set are updated
- If $m < 1$ or $n < 1$, an error occurs and the update fails
- If $n = m - 1$, no rows are updated
- If $n < m - 1$, an error occurs and the update fails

**ROWS Example**

```sql
UPDATE employees
SET salary = salary + 50
ORDER BY salary ASC
ROWS 20;
```

### 6.3.5. The RETURNING Clause

An UPDATE statement involving at most one row may include RETURNING in order to return some values from the row being updated. RETURNING may include data from any column of the row, not
necessarily the columns that are currently being updated. It can include literals or expressions not associated with columns, if there is a need for that.

When the RETURNING set contains data from the current row, the returned values report changes made in the BEFORE UPDATE triggers, but not those made in AFTER UPDATE triggers.

The context variables OLD.fieldname and NEW.fieldname can be used as column names. If OLD. or NEW. is not specified, the column values returned are the NEW. ones.

In DSQL, a statement with RETURNING always returns a single row. Attempts to execute an UPDATE … RETURNING … that affects multiple rows will result in the error “multiple rows in singleton select”. If the statement updates no records, the returned values contain NULL. This behaviour may change in future Firebird versions.

**The INTO Sub-clause**

In PSQL, the INTO clause can be used to pass the returning values to local variables. It is not available in DSQL. If no records are updated, nothing is returned and variables specified in RETURNING will keep their previous values.

**RETURNING Example (DSQL)**

```
UPDATE Scholars
SET firstname = 'Hugh', lastname = 'Pickering'
WHERE firstname = 'Henry' and lastname = 'Higgins'
RETURNING id, old.lastname, new.lastname;
```

**6.3.6. Updating BLOB columns**

Updating a BLOB column always replaces the entire contents. Even the BLOB ID, the “handle” that is stored directly in the column, is changed. BLOBs can be updated if:

1. The client application has made special provisions for this operation, using the Firebird API. In this case, the *modus operandi* is application-specific and outside the scope of this manual.
2. The new value is a string literal of no more than 65,533 bytes (64KB - 3).

A limit, in characters, is calculated at run-time for strings that are in multi-byte character sets, to avoid overrunning the bytes limit. For example, for a UTF8 string (max. 4 bytes/character), the run-time limit is likely to be about (floor(65533/4)) = 16383 characters.

3. The source is itself a BLOB column or, more generally, an expression that returns a BLOB.
4. You use the INSERT CURSOR statement (ESQL only).

**6.4. UPDATE OR INSERT**

*Used for*
Updating an existing record in a table or, if it does not exist, inserting it

Available in
DSQL, PSQL

Syntax

```
UPDATE OR INSERT INTO target [(<column_list>)]
VALUES (<value_list>)
[MATCHING (<column_list>)]
[RETURNING <value> [INTO <variables>]]
```

```
<column_list> ::= colname [, colname ...]

<value_list> ::= <value> [, <value> ...]

<returning_list> ::= <ret_value> [, <ret_value> ...]

<ret_value> ::=
    colname
    | NEW.colname
    | OLD.colname
    | <value>

<variables> ::= [:]varname [, [:]varname ...]
```

Table 82. Arguments for the UPDATE OR INSERT Statement Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>The name of the table or view where the record(s) is to be updated or a new record inserted</td>
</tr>
<tr>
<td>colname</td>
<td>Name of a column in the table or view</td>
</tr>
<tr>
<td>value</td>
<td>An expression whose value is to be used for inserting or updating the table, or returning a value</td>
</tr>
<tr>
<td>ret_value</td>
<td>An expression returned in the RETURNING clause</td>
</tr>
<tr>
<td>varname</td>
<td>Variable name — PSQL only</td>
</tr>
</tbody>
</table>

UPDATE OR INSERT inserts a new record or updates one or more existing records. The action taken depends on the values provided for the columns in the MATCHING clause (or, if the latter is absent, in the primary key). If there are records found matching those values, they are updated. If not, a new record is inserted. A match only counts if all the values in the MATCHING or primary key columns are equal. Matching is done with the \texttt{IS NOT DISTINCT} operator, so one NULL matches another.

Restrictions

- If the table has no primary key, the MATCHING clause is mandatory.
- In the MATCHING list as well as in the update/insert column list, each column
name may occur only once.

- The “INTO <variables>” subclause is only available in PSQL.
- When values are returned into the context variable NEW, this name must not be preceded by a colon (“:”).

6.4.1. The RETURNING clause

The optional RETURNING clause, if present, need not contain all the columns mentioned in the statement and may also contain other columns or expressions. The returned values reflect any changes that may have been made in BEFORE triggers, but not those in AFTER triggers. OLD.fieldname and NEW.fieldname may both be used in the list of columns to return; for field names not preceded by either of these, the new value is returned.

In DSQL, a statement with a RETURNING clause always returns exactly one row. If a RETURNING clause is present and more than one matching record is found, an error “multiple rows in singleton select” is raised. This behaviour may change in a later version of Firebird.

The optional INTO sub-clause is only valid in PSQL.

6.4.2. Example of UPDATE OR INSERT

Modifying data in a table, using UPDATE OR INSERT in a PSQL module. The return value is passed to a local variable, whose colon prefix is optional.

```sql
UPDATE OR INSERT INTO Cows (Name, Number, Location)
VALUES ('Suzy Creamcheese', 3278823, 'Green Pastures')
MATCHING (Number)
RETURNING rec_id into :id;
```

6.5. DELETE

Used for
Deleting rows from a table or view

Available in
DSQL, ESQL, PSQL

Syntax

```sql
DELETE
FROM target [[AS] alias]
[WHERE {<search-conditions> | CURRENT OF cursorname}]
[PLAN <plan_items>]
[ORDER BY <sort_items>]
[ROWS m [TO n]]
[RETURNING <returning_list> [INTO <variables>]]
```
Table 83. Arguments for the `DELETE` Statement Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>The name of the table or view from which the records are to be deleted</td>
</tr>
<tr>
<td>alias</td>
<td>Alias for the target table or view</td>
</tr>
<tr>
<td>search-conditions</td>
<td>Search condition limiting the set of rows being targeted for deletion</td>
</tr>
<tr>
<td>cursorname</td>
<td>The name of the cursor in which current record is positioned for deletion</td>
</tr>
<tr>
<td>plan_items</td>
<td>Query plan clause</td>
</tr>
<tr>
<td>sort_items</td>
<td>ORDER BY clause</td>
</tr>
<tr>
<td>m, n</td>
<td>Integer expressions for limiting the number of rows being deleted</td>
</tr>
<tr>
<td>ret_value</td>
<td>An expression to be returned in the <code>RETURNING</code> clause</td>
</tr>
<tr>
<td>value</td>
<td>An expression whose value is used for returning</td>
</tr>
<tr>
<td>varname</td>
<td>Name of a PSQL variable</td>
</tr>
</tbody>
</table>

DELETE removes rows from a database table or from one or more of the tables that underlie a view. WHERE and ROWS clauses can limit the number of rows deleted. If neither WHERE nor ROWS is present, DELETE removes all the rows in the relation.

### 6.5.1. Aliases

If an alias is specified for the target table or view, it must be used to qualify all field name references in the `DELETE` statement.

**Examples**

Supported usage:

```sql
delete from Cities where name starting 'Alex';
delete from Cities where Cities.name starting 'Alex';
delete from Cities C where name starting 'Alex';
delete from Cities C where C.name starting 'Alex';
```

Not possible:
6.5.2. WHERE

The WHERE clause sets the conditions that limit the set of records for a searched delete.

In PSQL, if a named cursor is being used for deleting a set, using the WHERE CURRENT OF clause, the action is limited to the row where the cursor is currently positioned. This is a positioned delete.

The WHERE CURRENT OF clause is available only in PSQL and ESQL, since there is no statement for creating and manipulating an explicit cursor in DSQL. Searched deletes are also available in PSQL, of course.

Examples

```
DELETE FROM People
WHERE firstname <> 'Boris' AND lastname <> 'Johnson';

DELETE FROM employee e
WHERE NOT EXISTS(
    SELECT *
    FROM employee_project ep
    WHERE e.emp_no = ep.emp_no);

DELETE FROM Cities
WHERE CURRENT OF Cur_Cities;  -- ESQL and PSQL only
```

6.5.3. PLAN

A PLAN clause allows the user to optimize the operation manually.

Example

```
DELETE FROM Submissions
WHERE date_entered < '1-Jan-2002'
PLAN (Submissions INDEX ix_subm_date);
```

6.5.4. ORDER BY and ROWS

The ORDER BY clause orders the set before the actual deletion takes place. It only makes sense in combination with ROWS, but is also valid without it.

The ROWS clause limits the number of rows being deleted. Integer literals or any integer expressions can be used for the arguments \( m \) and \( n \).

If ROWS has one argument, \( m \), the rows to be deleted will be limited to the first \( m \) rows.
Points to note

• If $m >$ the number of rows being processed, the entire set of rows is deleted
• If $m = 0,$ no rows are deleted
• If $m < 0,$ an error occurs and the deletion fails

If two arguments are used, $m$ and $n,$ ROWS limits the rows being deleted to rows from $m$ to $n$ inclusively. Both arguments are integers and start from 1.

Points to note

• If $m >$ the number of rows being processed, no rows are deleted
• If $m > 0$ and $\leq$ the number of rows in the set and $n$ is outside these values, rows from $m$ to the end of the set are deleted
• If $m < 1$ or $n < 1,$ an error occurs and the deletion fails
• If $n = m - 1,$ no rows are deleted
• If $n < m - 1,$ an error occurs and the deletion fails

Examples

Deleting the oldest purchase:

```
DELETE FROM Purchases
ORDER BY date ROWS 1;
```

Deleting the highest custno(s):

```
DELETE FROM Sales
ORDER BY custno DESC ROWS 1 to 10;
```

Deleting all sales, ORDER BY clause pointless:

```
DELETE FROM Sales
ORDER BY custno DESC;
```

Deleting one record starting from the end, i.e. from Z...:

```
DELETE FROM popgroups
ORDER BY name DESC ROWS 1;
```

Deleting the five oldest groups:

```
DELETE FROM popgroups
ORDER BY formed ROWS 5;
```
No sorting (ORDER BY) is specified so 8 found records, starting from the fifth one, will be deleted:

```sql
DELETE FROM popgroups
ROWS 5 TO 12;
```

### 6.5.5. RETURNING

A DELETE statement removing *at most one row* may optionally include a RETURNING clause in order to return values from the deleted row. The clause, if present, need not contain all the relation's columns and may also contain other columns or expressions.

- In DSQL, a statement with RETURNING always returns a singleton, never a multi-row set. If a RETURNING clause is present and more than one matching record is found, an error “multiple rows in singleton select” is raised. If no records are deleted, the returned columns contain NULL. This behaviour may change in future Firebird versions.

- The INTO clause is available only in PSQL
  - If the row is not deleted, nothing is returned and the target variables keep their values

#### Examples

```sql
DELETE FROM Scholars
WHERE firstname = 'Henry' and lastname = 'Higgins'
RETURNING lastname, fullname, id;
```

```sql
DELETE FROM Dumbbells
ORDER BY iq DESC
ROWS 1
RETURNING lastname, iq into :lname, :iq;
```

### 6.6. MERGE

*Used for*
Merging data from a source set into a target relation

*Available in*
DSQL, PSQL

*Syntax*

```sql
MERGE INTO target [[AS] target_alias]
USING <source> [[AS] source_alias]
ON <join_condition>
<merge_when> [<merge_when> ...]
```
[RETURNING <returning_list> [INTO <variables>]]

<merge_when> ::= 
  <merge_when_matched> 
  | <merge_when_not_matched>

<merge_when_matched> ::= 
  WHEN MATCHED [ AND <condition> ] THEN 
  { UPDATE SET <assignment-list> 
  | DELETE }

<merge_when_not_matched> ::= 
  WHEN NOT MATCHED [ AND <condition> ] THEN 
  INSERT [( <column_list> )] VALUES ( <value_list> )

<source> ::= tablename | (<select_stmt>)

<assignment_list> ::= 
  colname = <value> [, colname = <value> ...]

<column_list> ::= colname [, colname ...]

<value_list> ::= <value> [, <value> ...]

<returning_list> ::= 

<ret_value> ::= 
  colname 
  | table_or_alias.colname 
  | NEW.colname 
  | OLD.colname 
  | <value>

:variables ::= 
  [:]varname [, [:]varname ...]

Table 84. Arguments for the MERGE Statement Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>Name of target relation (table or updatable view)</td>
</tr>
<tr>
<td>source</td>
<td>Data source. It can be a table, a view, a stored procedure or a derived table</td>
</tr>
<tr>
<td>target_alias</td>
<td>Alias for the target relation (table or updatable view)</td>
</tr>
<tr>
<td>source_alias</td>
<td>Alias for the source relation or set</td>
</tr>
<tr>
<td>join_conditions</td>
<td>The (ON) condition(s) for matching the source records with those in the target</td>
</tr>
<tr>
<td>condition</td>
<td>Additional test condition in WHEN MATCHED or WHEN NOT MATCHED clause</td>
</tr>
</tbody>
</table>
### Argument | Description
--- | ---
tablename | Table or view name
select_stmt | Select statement of the derived table
colname | Name of a column in the target relation
value | The value assigned to a column in the target table. This expression may be a literal value, a PSQL variable, a column from the source, or a compatible context variable
ret_value | The expression to be returned in the RETURNING clause. Can be a column reference to source or target, or a column reference of the NEW or OLD context of the target, or a value.
ret_alias | Alias for the value expression in the RETURNING clause
varname | Name of a PSQL local variable

The `MERGE` statement merges records from the source into a target table or updatable view. The source may be a table, view or “anything you can SELECT from” in general. Each source record will be used to update one or more target records, insert a new record in the target table, delete a record from the target table or do nothing.

The action taken depends on the supplied join condition, the `WHEN` clause(s), and the - optional - condition in the `WHEN` clause. The join condition and condition in the `WHEN` will typically contain a comparison of fields in the source and target relations.

Multiple `WHEN MATCHED` and `WHEN NOT MATCHED` clauses are allowed. For each row in the source, the `WHEN` clauses are checked in the order they are specified in the statement. If the condition in the `WHEN` clause does not evaluate to true, the clause is skipped, and the next clause will be checked. This will be done until the condition for a `WHEN` clause evaluates to true, or a `WHEN` clauses without condition matches, or there are no more `WHEN` clauses. If a matching clause is found, the action associated with the clause is executed. For each row in the source, at most one action is executed.

At least one `WHEN` clause must be present.

`WHEN NOT MATCHED` is evaluated from the source viewpoint, that is, the table or set specified in `USING`. It has to work this way because if the source record does not match a target record, `INSERT` is executed. Of course, if there is a target record which does not match a source record, nothing is done.

Currently, the `ROW_COUNT` variable returns the value 1, even if more than one record is modified or inserted. For details and progress, refer to Tracker ticket CORE-4400.

**ALERT : Another irregularity!**

If the `WHEN MATCHED` clause is present and several records match a single record in the target table, an `UPDATE` will be executed on that one target record for each one of the matching source records, with each successive update overwriting the previous one. This behaviour does not comply with the SQL:2003 standard, which requires that this situation throw an exception (an error).
This has been fixed in Firebird 4, and will raise an error instead. See also CORE-2274

6.6.1. The RETURNING Clause

A MERGE statement that affects at most one row can contain a RETURNING clause to return values added, modified or removed. If a RETURNING clause is present and more than one matching record is found, an error “multiple rows in singleton select” is raised. The RETURNING clause can contain any columns from the target table (or updateable view), as well as other columns (eg from the source) and expressions.

The optional INTO sub-clause is only valid in PSQL.

The restriction that RETURNING can only be used with a statement that affects at most one row might be removed in a future version.

Column names can be qualified by the OLD or NEW prefix to define exactly what value to return: before or after modification. The returned values include the changes made by BEFORE triggers.

For the UPDATE or INSERT action, unqualified column names or those qualified by the target table name or alias will behave as if qualified by NEW, while for the DELETE action as if qualified by OLD.

The following example modifies the previous example to affect one line, and adds a RETURNING clause to return the old and new quantity of goods, and the difference between those values.

Using MERGE with a RETURNING clause

```sql
MERGE INTO PRODUCT_INVENTORY AS TARGET
USING (SELECT
    SL.ID_PRODUCT,
    SUM(SL.QUANTITY)
FROM SALES_ORDER_LINE SL
JOIN SALES_ORDER S ON S.ID = SL.ID_SALES_ORDER
WHERE S.BYDATE = CURRENT_DATE
AND SL.ID_PRODUCT =: ID_PRODUCT
GROUP BY 1 ) AS SRC (ID_PRODUCT, QUANTITY)
ON TARGET.ID_PRODUCT = SRC.ID_PRODUCT
WHEN MATCHED AND TARGET.QUANTITY - SRC.QUANTITY <= 0 THEN
    DELETE
WHEN MATCHED THEN
    UPDATE SET
        TARGET.QUANTITY = TARGET.QUANTITY - SRC.QUANTITY,
        TARGET.BYDATE = CURRENT_DATE
RETURNING OLD.QUANTITY, NEW.QUANTITY, SRC.QUANTITY
INTO : OLD_QUANTITY, :NEW_QUANTITY, :DIFF_QUANTITY
```
6.6.2. Examples of **MERGE**

1. Update books when present, or add new record if absent

   ```sql
   MERGE INTO books b
   USING purchases p
   ON p.title = b.title and p.type = 'bk'
   WHEN MATCHED THEN
     UPDATE SET b.desc = b.desc || '; ' || p.desc
   WHEN NOT MATCHED THEN
     INSERT (title, desc, bought) values (p.title, p.desc, p.bought);
   ```

2. Using a derived table

   ```sql
   MERGE INTO customers c
   USING (SELECT * from customers_delta WHERE id > 10) cd
   ON (c.id = cd.id)
   WHEN MATCHED THEN
     UPDATE SET name = cd.name
   WHEN NOT MATCHED THEN
     INSERT (id, name) values (cd.id, cd.name);
   ```

3. Together with a recursive CTE

   ```sql
   MERGE INTO numbers
   USING (WITH RECURSIVE r(n) AS (
     SELECT 1 FROM rdb$database
     UNION ALL
     SELECT n+1 FROM r WHERE n < 200
   )
   SELECT n FROM r)
   t
   ON numbers.num = t.n
   WHEN NOT MATCHED THEN
     INSERT(num) VALUES(t.n);
   ```

4. Using **DELETE** clause

   ```sql
   MERGE INTO SALARY_HISTORY
   USING (SELECT EMP_NO FROM EMPLOYEE
     WHERE DEPT_NO = 120) EMP
   ON SALARY_HISTORY.EMP_NO = EMP.EMP_NO
   WHEN MATCHED THEN DELETE
   ```
5. The following example updates the `PRODUCT_INVENTORY` table daily based on orders processed in the `SALES_ORDER_LINE` table. If the stock level of the product would drop to zero or lower, then the row for that product is removed from the `PRODUCT_INVENTORY` table.

```
MERGE INTO PRODUCT_INVENTORY AS TARGET
USING ( 
    SELECT 
        SL.ID_PRODUCT, 
        SUM (SL.QUANTITY) 
    FROM SALES_ORDER_LINE SL 
    JOIN SALES_ORDER S ON S.ID = SL.ID_SALES_ORDER 
    WHERE S.BYDATE = CURRENT_DATE 
    GROUP BY 1 
) AS SRC (ID_PRODUCT, QUANTITY) 
ON TARGET.ID_PRODUCT = SRC.ID_PRODUCT 
WHEN MATCHED AND TARGET.QUANTITY - SRC.QUANTITY <= 0 THEN 
    DELETE 
WHEN MATCHED THEN 
    UPDATE SET 
        TARGET.QUANTITY = TARGET.QUANTITY - SRC.QUANTITY, 
        TARGET.BYDATE = CURRENT_DATE 
```

See also

`SELECT`, `INSERT`, `UPDATE`, `UPDATE OR INSERT`, `DELETE`

### 6.7. EXECUTE PROCEDURE

*Used for*

Executing a stored procedure

*Available in*

DSQL, ESQL, PSQL

*Syntax*

```
EXECUTE PROCEDURE procname 
    [{ <inparam-list | ( <inparam-list> ) }] 
    [RETURNING_VALUES { <outvar-list> | ( <outvar-list> ) }]

<inparam-list> ::= 
    <inparam> [, <inparam> ...]

<outvar-list> ::= 
    <outvar> [, <outvar> ...]

<outvar> ::= [:]varname
```

*Table 85. Arguments for the EXECUTE PROCEDURE Statement Parameters*
EXECUTES AN \textit{EXECUTABLE STORED PROCEDURE}, taking a list of one or more input parameters, if they are defined for the procedure, and returning a one-row set of output values, if they are defined for the procedure.

\textbf{6.7.1. “Executable” Stored Procedure}

The \texttt{EXECUTE PROCEDURE} statement is most commonly used to invoke the style of stored procedure that is written to perform some data-modifying task at the server side—those that do not contain any \texttt{SUSPEND} statements in their code. They can be designed to return a result set, consisting of only one row, which is usually passed, via a set of \texttt{RETURNING VALUES()} variables, to another stored procedure that calls it. Client interfaces usually have an API wrapper that can retrieve the output values into a single-row buffer when calling \texttt{EXECUTE PROCEDURE} in DSQL.

Invoking the other style of stored procedure—a “selectable” one—is possible with \texttt{EXECUTE PROCEDURE}, but it returns only the first row of an output set which is almost surely designed to be multi-row. Selectable stored procedures are designed to be invoked by a \texttt{SELECT} statement, producing output that behaves like a virtual table.

- In PSQL and DSQL, input parameters may be any expression that resolves to the expected type.
- Although parentheses are not required after the name of the stored procedure to enclose the input parameters, their use is recommended for the sake of good housekeeping.
- Where output parameters have been defined in a procedure, the \texttt{RETURNING VALUES} clause can be used in PSQL to retrieve them into a list of previously declared variables that conforms in sequence, data type and number with the defined output parameters.
- The list of \texttt{RETURNING VALUES} may be optionally enclosed in parentheses and their use is recommended.
- When DSQL applications call \texttt{EXECUTE PROCEDURE} using the Firebird API or some form of wrapper for it, a buffer is prepared to receive the output row and the \texttt{RETURNING VALUES} clause is not used.

\textbf{6.7.2. Examples of \texttt{EXECUTE PROCEDURE}}

1. In PSQL, with optional colons and without optional parentheses:

\begin{verbatim}
EXECUTE PROCEDURE MakeFullName
   :FirstName, :MiddleName, :LastName
\end{verbatim}
2. In Firebird’s command-line utility isql, with literal parameters and optional parentheses:

```sql
EXECUTE PROCEDURE MakeFullName ('J', 'Edgar', 'Hoover');
```

In DSQL (eg in isql), RETURNING_VALUES is not used. Any output values are captured by the application and displayed automatically.

3. A PSQL example with expression parameters and optional parentheses:

```sql
EXECUTE PROCEDURE MakeFullName ('Mr./Mrs. ' || FirstName, MiddleName, upper(LastName)) RETURNING_VALUES (FullName);
```

### 6.8. EXECUTE BLOCK

*Used for*

Creating an “anonymous” block of PSQL code in DSQL for immediate execution

*Available in*

DSQL

*Syntax*

```sql
EXECUTE BLOCK [(<inparams>)]
[RETURNS (<outparams>)]
<psql-module-body>
```

- `<inparams> ::=`  
  `<param_decl> = ? [, <inparams> ]`

- `<outparams> ::=`  
  `<param_decl> [, <outparams>]`

- `<param_decl> ::=`  
  `paramname <domain_or_non_array_type> [NOT NULL] [COLLATE collation]`

- `<domain_or_non_array_type> ::=`  
  `!! See Scalar Data Types Syntax !!`

- `<psql-module-body> ::=`  
  `!! See Syntax of a Module Body !!`

*Table 86. Arguments for the EXECUTE BLOCK Statement Parameters*

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>param_decl</td>
<td>Name and description of an input or output parameter</td>
</tr>
<tr>
<td>Argument</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>paramname</td>
<td>The name of an input or output parameter of the procedural block, up to 31 characters long. The name must be unique among input and output parameters and local variables in the block.</td>
</tr>
<tr>
<td>collation</td>
<td>Collation sequence</td>
</tr>
</tbody>
</table>

Executes a block of PSQL code as if it were a stored procedure, optionally with input and output parameters and variable declarations. This allows the user to perform “on-the-fly” PSQL within a DSQL context.

### 6.8.1. Examples

1. This example injects the numbers 0 through 127 and their corresponding ASCII characters into the table ASCIITABLE:

```sql
EXECUTE BLOCK
AS
    declare i INT = 0;
BEGIN
    WHILE (i < 128) DO
        BEGIN
            INSERT INTO AsciiTable VALUES (:i, ascii_char(:i));
            i = i + 1;
        END
    END
END
```

2. The next example calculates the geometric mean of two numbers and returns it to the user:

```sql
EXECUTE BLOCK (x DOUBLE PRECISION = ?, y DOUBLE PRECISION = ?)
RETURNS (gmean DOUBLE PRECISION)
AS
BEGIN
    gmean = SQRT(x*y);
    SUSPEND;
END
```

Because this block has input parameters, it has to be prepared first. Then the parameters can be set and the block executed. It depends on the client software how this must be done and even if it is possible at all — see the notes below.

3. Our last example takes two integer values, smallest and largest. For all the numbers in the range smallest...largest, the block outputs the number itself, its square, its cube and its fourth power.

```sql
EXECUTE BLOCK (smallest INT = ?, largest INT = ?)
RETURNS (number INT, square BIGINT, cube BIGINT, fourth BIGINT)
```
BEGIN
    number = smallest;
    WHILE (number <= largest) DO
        BEGIN
            square = number * number;
            cube = number * square;
            fourth = number * cube;
            SUSPEND;
            number = number + 1;
        END
    END

Again, it depends on the client software if and how you can set the parameter values.

### 6.8.2. Input and output parameters

Executing a block without input parameters should be possible with every Firebird client that allows the user to enter his or her own DSQL statements. If there are input parameters, things get trickier: these parameters must get their values after the statement is prepared, but before it is executed. This requires special provisions, which not every client application offers. (Firebird’s own isql, for one, doesn’t.)

The server only accepts question marks (“?”) as placeholders for the input values, not “:a”, “:MyParam” etc., or literal values. Client software may support the “:xxx” form though, and will preprocess it before sending it to the server.

If the block has output parameters, you must use SUSPEND or nothing will be returned.

Output is always returned in the form of a result set, just as with a SELECT statement. You can’t use RETURNING_VALUES or execute the block INTO some variables, even if there is only one result row.

#### PSQL Links

For more information about writing PSQL, consult Chapter *Procedural SQL (PSQL) Statements*.

### 6.8.3. Statement Terminators

Some SQL statement editors—specifically the isql utility that comes with Firebird and possibly some third-party editors—employ an internal convention that requires all statements to be terminated with a semi-colon. This creates a conflict with PSQL syntax when coding in these environments. If you are unacquainted with this problem and its solution, please study the details in the PSQL chapter in the section entitled *Switching the Terminator in isql*. 
Chapter 7. Procedural SQL (PSQL) Statements

Procedural SQL (PSQL) is a procedural extension of SQL. This language subset is used for writing stored procedures, triggers, and PSQL blocks.

PSQL provides all the basic constructs of traditional structured programming languages, and also includes DML statements (SELECT, INSERT, UPDATE, DELETE, etc.), with a slight modified syntax in some cases.

7.1. Elements of PSQL

A procedural extension may contain declarations of local variables, routines and cursors, assignments, conditional statements, loops, statements for raising custom exceptions, error handling and sending messages (events) to client applications. Triggers have access to special context variables, two arrays that store, respectively, the NEW values for all columns during insert and update activity, and the OLD values during update and delete work.

Statements that modify metadata (DDL) are not available in PSQL.

7.1.1. DML Statements with Parameters

If DML statements (SELECT, INSERT, UPDATE, DELETE, etc.) in the body of the module (procedure, function, trigger or block) use parameters, only named parameters can be used. If DML statements contain named parameters, then they must be previously declared as local variables using DECLARE [VARIABLE] in the declaration section of the module, or as input or output variables in the module header.

When a DML statement with parameters is included in PSQL code, the parameter name must be prefixed by a colon (‘:') in most situations. The colon is optional in statement syntax that is specific to PSQL, such as assignments and conditionals and the INTO clause. The colon prefix on parameters is not required when calling stored procedures from within another PSQL module or in DSQL.

7.1.2. Transactions

Stored procedures and functions (including those defined in packages) are executed in the context of the transaction in which they are called. Triggers are executed as an intrinsic part of the operation of the DML statement: thus, their execution is within the same transaction context as the statement itself. Individual transactions are launched for database event triggers.

Statements that start and end transactions are not available in PSQL, but it is possible to run a statement or a block of statements in an autonomous transaction.

7.1.3. Module Structure

PSQL code modules consist of a header and a body. The DDL statements for defining them are complex statements; that is, they consist of a single statement that encloses blocks of multiple statements. These statements begin with a verb (CREATE, ALTER, DROP, RECREATE, CREATE OR ALTER) and end with the last END statement of the body.
The Module Header

The header provides the module name and defines any input and output parameters or—for functions—the return type. Stored procedures and PSQL blocks may have input and output parameters. Functions may have input parameters and must have a scalar return type. Triggers do not have either input or output parameters.

The header of a trigger indicates the database event (insert, update or delete, or a combination) and the phase of operation (BEFORE or AFTER that event) that will cause it to “fire”.

The Module Body

The module body is either a PSQL module body, or an external module body.

Syntax of a Module Body

```
<module-body> ::=  
    <psql-module-body> | <external-module-body>

<psql-module-body> ::=  
    AS  
    [<declarations>]  
    BEGIN  
    [<PSQL_statements>]  
    END

<external-module-body> ::=  
    EXTERNAL [NAME <extname>] ENGINE engine  
    [AS '<extbody>']

<declarations> ::= <declare-item> [<declare-item ...]  

<declare-item> ::=  
    <declare-var>  
    | <declare-cursor>  
    | <declare-subfunc>  
    | <declare-subproc>

<extname> ::=  
    '<module-name>!<routine-name>![<misc-info>]'  

Table 87. Module Body Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>declarations</td>
<td>Section for declaring local variables, named cursors, and subroutines</td>
</tr>
<tr>
<td>PSQL_statements</td>
<td>Procedural SQL statements. Some PSQL statements may not be valid in all types of PSQL. For example, RETURN &lt;value&gt;; is only valid in functions.</td>
</tr>
<tr>
<td>declare_var</td>
<td>Local variable declaration</td>
</tr>
<tr>
<td>declare_cursor</td>
<td>Named cursor declaration</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>declare-subfunc</td>
<td>Sub-function declaration</td>
</tr>
<tr>
<td>declare-subproc</td>
<td>Sub-procedure declaration</td>
</tr>
<tr>
<td>extname</td>
<td>String identifying the external procedure</td>
</tr>
<tr>
<td>engine</td>
<td>String identifying the UDR engine</td>
</tr>
<tr>
<td>extbody</td>
<td>External procedure body. A string literal that can be used by UDRs for various purposes.</td>
</tr>
<tr>
<td>module-name</td>
<td>The name of the module that contains the procedure</td>
</tr>
<tr>
<td>routine-name</td>
<td>The internal name of the procedure inside the external module</td>
</tr>
<tr>
<td>misc-info</td>
<td>Optional string that is passed to the procedure in the external module</td>
</tr>
</tbody>
</table>

**The PSQL Module Body**

The PSQL body starts with an optional section that declares variables and subroutines, followed by a block of statements that run in a logical sequence, like a program. A block of statements — or compound statement — is enclosed by the `BEGIN` and `END` keywords, and is executed as a single unit of code. The main `BEGIN`…`END` block may contain any number of other `BEGIN`…`END` blocks, both embedded and sequential. Blocks can be nested to a maximum depth of 512 blocks. All statements except `BEGIN` and `END` are terminated by semicolons (';'). No other character is valid for use as a terminator for PSQL statements.

In the Firebird 2.5 Language Reference, the declaration of local variables and cursors was considered part of the module header. With the introduction of UDR (external routines) in Firebird 3.0, we now consider this declaration section part of the — PSQL — module body.

**Switching the Terminator in isql**

Here we digress a little, to explain how to switch the terminator character in the `isql` utility to make it possible to define PSQL modules in that environment without conflicting with `isql` itself, which uses the same character, semicolon (';'), as its own statement terminator.

**isql Command**

**SET TERM**

*Used for*

Changing the terminator character(s) to avoid conflict with the terminator character in PSQL statements

*Available in*

ISQL only

*Syntax*

```
SET TERM new_terminator old_terminator
```
Table 88. SET TERM Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_terminator</td>
<td>New terminator</td>
</tr>
<tr>
<td>old_terminator</td>
<td>Old terminator</td>
</tr>
</tbody>
</table>

When you write your triggers and stored procedures in `isql`—either in the interactive interface or in scripts—running a `SET TERM` statement is needed to switch the normal `isql` statement terminator from the semicolon to some other character or short string, to avoid conflict with the non-changeable semicolon terminator in PSQL. The switch to an alternative terminator needs to be done before you begin defining PSQL objects or running your scripts.

The alternative terminator can be any string of characters except for a space, an apostrophe or the current terminator character(s). Any letter character(s) used will be case-sensitive.

**Example**

Changing the default semicolon to `^` (caret) and using it to submit a stored procedure definition: character as an alternative terminator character:

```sql
SET TERM ^;
CREATE OR ALTER PROCEDURE SHIP_ORDER (PO_NUM CHAR(8))
AS BEGIN
  /* Stored procedure body */
END^
/* Other stored procedures and triggers */
SET TERM ;^ /* Other DDL statements */
```

**The External Module Body**

The external module body specifies the UDR engine used to execute the external module, and optionally specifies the name of the UDR routine to call (`<extname>`) and/or a string (`<extbody>`) with UDR-specific semantics.

Configuration of external modules and UDR engines is not covered further in this Language Reference. Consult the documentation of a specific UDR engine for details.

**7.2. Stored Procedures**

A stored procedure is executable code stored in the database metadata for execution on the server. A stored procedure can be called by other stored procedures (including itself), functions, triggers...
and client applications. A procedure that calls itself is known as recursive.

### 7.2.1. Benefits of Stored Procedures

Stored procedures have the following advantages:

**Modularity**

Applications working with the database can use the same stored procedure, thereby reducing the size of the application code and avoiding code duplication.

**Simpler Application Support**

When a stored procedure is modified, changes appear immediately to all host applications, without the need to recompile them if the parameters were unchanged.

**Enhanced Performance**

Since stored procedures are executed on a server instead of at the client, network traffic is reduced, which improves performance.

### 7.2.2. Types of Stored Procedures

Firebird supports two types of stored procedures: *executable* and *selectable*.

**Executable Procedures**

Executable procedures usually modify data in a database. They can receive input parameters and return a single set of output (RETURNS) parameters. They are called using the `EXECUTE PROCEDURE` statement. See an example of an executable stored procedure at the end of the `CREATE PROCEDURE` section of Chapter 5.

**Selectable Procedures**

Selectable stored procedures usually retrieve data from a database, returning an arbitrary number of rows to the caller. The caller receives the output one row at a time from a row buffer that the database engine prepares for it.

Selectable procedures can be useful for obtaining complex sets of data that are often impossible or too difficult or too slow to retrieve using regular DSQL `SELECT` queries. Typically, this style of procedure iterates through a looping process of extracting data, perhaps transforming it before filling the output variables (parameters) with fresh data at each iteration of the loop. A `SUSPEND` statement at the end of the iteration fills the buffer and waits for the caller to fetch the row. Execution of the next iteration of the loop begins when the buffer has been cleared.

Selectable procedures may have input parameters, and the output set is specified by the `RETURNS` clause in the header.

A selectable stored procedure is called with a `SELECT` statement. See an example of a selectable stored procedure at the end of the `CREATE PROCEDURE` section of Chapter 5.
7.2.3. Creating a Stored Procedure

The syntax for creating executable stored procedures and selectable stored procedures is exactly the same. The difference comes in the logic of the program code.

For information about creating stored procedures, see CREATE PROCEDURE in Chapter Data Definition (DDL) Statements.

7.2.4. Modifying a Stored Procedure

For information about modifying existing stored procedures, see ALTER PROCEDURE, CREATE OR ALTER PROCEDURE, RECREATE PROCEDURE, in Chapter Data Definition (DDL) Statements.

7.2.5. Deleting a Stored Procedure

For information about deleting stored procedures, see DROP PROCEDURE in Chapter Data Definition (DDL) Statements.

7.3. Stored Functions

A stored function is executable code stored in the database metadata for execution on the server. A stored function can be called by other stored functions (including itself), procedures, triggers and client applications. A function that calls itself is known as recursive.

Unlike stored procedures, stored functions always return one scalar value. To return a value from a stored function, use the RETURN statement, which immediately terminates the function.

7.3.1. Creating a Stored Function

For information about creating stored functions, see CREATE FUNCTION in Chapter Data Definition (DDL) Statements.

7.3.2. Modifying a Stored Function

For information about modifying stored functions, see ALTER FUNCTION, CREATE OR ALTER FUNCTION, RECREATE FUNCTION, in Chapter Data Definition (DDL) Statements.

7.3.3. Deleting a Stored Function

For information about deleting stored procedures, see DROP FUNCTION in Chapter Data Definition (DDL) Statements.

7.4. PSQL Blocks

A self-contained, unnamed (“anonymous”) block of PSQL code can be executed dynamically in DSQL, using the EXECUTE BLOCK syntax. The header of an anonymous PSQL block may optionally contain input and output parameters. The body may contain local variables, cursor declarations and local routines, followed by a block of PSQL statements.
An anonymous PSQL block is not defined and stored as an object, unlike stored procedures and triggers. It executes in run-time and cannot reference itself.

Just like stored procedures, anonymous PSQL blocks can be used to process data and to retrieve data from the database.

**Syntax (incomplete)**

```sql
EXECUTE BLOCK
  [([<inparam> = ? [, <inparam> = ? ...]])
  [RETURNS (<outparam> [, <outparam> ...])]]
<psql-module-body>

<psql-module-body> ::= 
  !! See Syntax of Module Body !!
```

**Table 89. PSQL Block Parameters**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inparam</td>
<td>Input parameter description</td>
</tr>
<tr>
<td>outparam</td>
<td>Output parameter description</td>
</tr>
<tr>
<td>declarations</td>
<td>A section for declaring local variables and named cursors</td>
</tr>
<tr>
<td>PSQL statements</td>
<td>PSQL and DML statements</td>
</tr>
</tbody>
</table>

See also

See `EXECUTE BLOCK` for details.

### 7.5. Packages

A package is a group of stored procedures and function defined as a single database object.

Firebird packages are made up of two parts: a header (`PACKAGE` keyword) and a body (`PACKAGE BODY` keywords). This separation is very similar to Delphi modules, the header corresponds to the interface part, and the body corresponds to the implementation part.

#### 7.5.1. Benefits of Packages

The notion of “packaging” the code components of a database operation addresses has several advantages:

**Modularisation**

Blocks of interdependent code are grouped into logical modules, as done in other programming languages.

In programming, it is well recognised that grouping code in various ways, in namespaces, units or classes, for example, is a good thing. This is not possible with standard stored procedures and functions in the database. Although they can be grouped in different script files, two problems
remain:

a. The grouping is not represented in the database metadata.
b. Scripted routines all participate in a flat namespace and are callable by everyone (we are not referring to security permissions here).

Easier tracking of dependencies

Packages make it easy to track dependencies between a collection of related routines, as well as between this collection and other routines, both packaged and unpackaged.

Whenever a packaged routine determines that it uses a certain database object, a dependency on that object is registered in Firebird’s system tables. Thereafter, to drop, or maybe alter that object, you first need to remove what depends on it. Since the dependency on other objects only exists for the package body, and not the package body, this package body can easily be removed, even if some other object depends on this package. When the body is dropped, the header remains, allowing you to recreate its body once the changes related to the removed object are done.

Simplify permission management

As Firebird runs routines with the caller privileges, it is necessary also to grant resource usage to each routine when these resources would not be directly accessible to the caller. Usage of each routine needs to be granted to users and/or roles.

Packaged routines do not have individual privileges. The privileges apply to the package as a whole. Privileges granted to packages are valid for all package body routines, including private ones, but are stored for the package header. An EXECUTE privilege on a package granted to a user (or other object), grants that user the privilege to execute all routines defined in the package header.

For example

```
GRANT SELECT ON TABLE secret TO PACKAGE pk_secret;
GRANT EXECUTE ON PACKAGE pk_secret TO ROLE role_secret;
```

Private scopes

Stored procedures and functions can be privates; that is, make them available only for internal usage within the defining package.

All programming languages have the notion of routine scope, which is not possible without some form of grouping. Firebird packages also work like Delphi units in this regard. If a routine is not declared in the package header (interface) and is implemented in the body (implementation), it becomes a private routine. A private routine can only be called from inside its package.

7.5.2. Creating a Package

For information on creating packages, see CREATE PACKAGE, CREATE PACKAGE BODY
7.5.3. Modifying a Package

For information on modifying existing package header or bodies, see ALTER PACKAGE, CREATE OR ALTER PACKAGE, RECREATE PACKAGE, RECREATE PACKAGE BODY

7.5.4. Deleting a Package

For information on deleting a package, see DROP PACKAGE, DROP PACKAGE BODY

7.6. Triggers

A trigger is another form of executable code that is stored in the metadata of the database for execution by the server. A trigger cannot be called directly. It is called automatically (“fired”) when data-changing events involving one particular table or view occur, or on a specific database event.

A trigger applies to exactly one table or view or database event, and only one phase in an event (BEFORE or AFTER the event). A single DML trigger might be written to fire only when one specific data-changing event occurs (INSERT, UPDATE or DELETE), or it might be written to apply to more than one of those.

A DML trigger is executed in the context of the transaction in which the data-changing DML statement is running. For triggers that respond to database events, the rule is different: for DDL triggers and transaction triggers, the trigger runs in the same transaction that executed the DDL, for other types, a new default transaction is started.

7.6.1. Firing Order (Order of Execution)

More than one trigger can be defined for each phase-event combination. The order in which they are executed (known as “firing order” can be specified explicitly with the optional POSITION argument in the trigger definition. You have 32,767 numbers to choose from. Triggers with the lowest position numbers fire first.

If a POSITION clause is omitted, or if several matching event-phase triggers have the same position number, then the triggers will fire in alphabetical order.

7.6.2. DML Triggers

DML triggers are those that fire when a DML operation changes the state of data: updating rows in tables, inserting new rows or deleting rows. They can be defined for both tables and views.

Trigger Options

Six base options are available for the event-phase combination for tables and views:

- Before a new row is inserted: BEFORE INSERT
- After a new row is inserted: AFTER INSERT
- Before a row is updated: BEFORE UPDATE
- After a row is updated: AFTER UPDATE
Before a row is deleted
BEFORE DELETE

After a row is deleted
AFTER DELETE

These base forms are for creating single phase/single-event triggers. Firebird also supports forms for creating triggers for one phase and multiple-events, BEFORE INSERT OR UPDATE OR DELETE, for example, or AFTER UPDATE OR DELETE: the combinations are your choice.

“Multi-phase” triggers, such as BEFORE OR AFTER …, are not possible.

The Boolean context variables INSERTING, UPDATING and DELETING can be used in the body of a trigger to determine the type of event that fired the trigger.

OLD and NEW Context Variables

For DML triggers, the Firebird engine provides access to sets of OLD and NEW context variables. Each is an array of the values of the entire row: one for the values as they are before the data-changing event (the BEFORE phase) and one for the values as they will be after the event (the AFTER phase). They are referenced in statements using the form NEW.column_name and OLD.column_name, respectively. The column_name can be any column in the table’s definition, not just those that are being updated.

The NEW and OLD variables are subject to some rules:

- In all triggers, the OLD value is read-only
- In BEFORE UPDATE and BEFORE INSERT code, the NEW value is read/write, unless it is a COMPUTED BY column
- In INSERT triggers, references to the OLD variables are invalid and will throw an exception
- In DELETE triggers, references to the NEW variables are invalid and will throw an exception
- In all AFTER trigger code, the NEW variables are read-only

7.6.3. Database Triggers

A trigger associated with a database or transaction event can be defined for the following events:

Connecting to a database
ON CONNECT
Before the trigger is executed, a transaction is automatically started with the default isolation level (snapshot concurrency), write, wait

Disconnecting from a database
ON DISCONNECT
Before the trigger is executed, a transaction is automatically started with the default isolation level (snapshot concurrency), write, wait

When a transaction is started
ON TRANSACTION START
The trigger is executed in the current transaction context

When a transaction is committed
ON TRANSACTION COMMIT
The trigger is executed in the current transaction context
When a transaction is cancelled, the trigger is executed in the current transaction context.

### 7.6.4. DDL Triggers

DDL triggers fire on specified metadata changes events in a specified phase. **BEFORE** triggers run before changes to system tables. **AFTER** triggers run after changes in system tables.

DDL triggers are a specific type of database trigger, so most rules for and semantics of database triggers also apply for DDL triggers.

**Semantics**

1. **BEFORE** triggers are fired before changes to the system tables. **AFTER** triggers are fired after system table changes.

   **Important Rule**

   The event type [**BEFORE** | **AFTER**] of a DDL trigger cannot be changed.

2. When a DDL statement fires a trigger that raises an exception (**BEFORE** or **AFTER**, intentionally or unintentionally) the statement will not be committed. That is, exceptions can be used to ensure that a DDL operation will fail if the conditions are not precisely as intended.

3. DDL trigger actions are executed only when **committing** the transaction in which the affected DDL command runs. Never overlook the fact that what is possible to do in an **AFTER** trigger is exactly what is possible to do after a DDL command without autocommit. You cannot, for example, create a table and then use it in the trigger.

4. With “**CREATE** OR **ALTER**” statements, a trigger is fired one time at the **CREATE** event or the **ALTER** event, according to the previous existence of the object. With **RECREATE** statements, a trigger is fired for the **DROP** event if the object exists, and for the **CREATE** event.

5. **ALTER** and **DROP** events are generally not fired when the object name does not exist. For the exception, see point 6.

6. The exception to rule 5 is that **BEFORE ALTER/DROP USER** triggers fire even when the username does not exist. This is because, underneath, these commands perform DML on the security database, and the verification is not done before the command on it is run. This is likely to be different with embedded users, so do not write code that depends on this.

7. If some exception is raised after the DDL command starts its execution and before **AFTER** triggers are fired, **AFTER** triggers will not be fired.

8. Packaged procedures and triggers do not fire individual {**CREATE** | **ALTER** | **DROP**} {**PROCEDURE** | **FUNCTION**} triggers.

**The DDL_TRIGGER Context Namespace**

When a DDL trigger is running, the **DDL_TRIGGER** namespace is available for use with **RDB$GET_CONTEXT**. This namespace contains information on the currently firing trigger.

See also **The DDL_TRIGGER Namespace** in **RDB$GET_CONTEXT** in Chapter *Built-in Scalar Functions*. 

325
7.6.5. Creating Triggers

For information on creating triggers, see `CREATE TRIGGER`, `CREATE OR ALTER TRIGGER`, `RECREATE TRIGGER` in Chapter *Data Definition (DDL) Statements*.

7.6.6. Modifying Triggers

For information on modifying triggers, see `ALTER TRIGGER`, `CREATE OR ALTER TRIGGER`, `RECREATE TRIGGER` in Chapter *Data Definition (DDL) Statements*.

7.6.7. Deleting a Trigger

For information on deleting triggers, see `DROP TRIGGER` in Chapter *Data Definition (DDL) Statements*.

7.7. Writing the Body Code

This section takes a closer look at the procedural SQL language constructs and statements that are available for coding the body of a stored procedure, trigger or anonymous PSQL block.

---

### Colon Marker (’:’)

The colon marker prefix (‘:’) is used in PSQL to mark a reference to a variable in a DML statement. The colon marker is not required before variable names in other PSQL code.

Since Firebird 3.0, the colon prefix can also be used for the `NEW` and `OLD` contexts, and for cursor variables.

---

#### 7.7.1. Assignment Statements

*Used for*

Assigning a value to a variable

*Available in*

PSQL

*Syntax*

```plaintext
varname = <value_expr>;
```

*Table 90. Assignment Statement Parameters*

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varname</td>
<td>Name of a parameter or local variable</td>
</tr>
<tr>
<td>value_expr</td>
<td>An expression, constant or variable whose value resolves to the same data type as varname</td>
</tr>
</tbody>
</table>

PSQL uses the equal symbol (‘=’) as its assignment operator. The assignment statement assigns an
SQL expression value on the right to the variable on the left of the operator. The expression can be any valid SQL expression: it may contain literals, internal variable names, arithmetic, logical and string operations, calls to internal functions, stored functions or external functions (UDFs).

### Example using assignment statements

```
CREATE PROCEDURE MYPROC (  
    a INTEGER,
    b INTEGER,
    name VARCHAR (30)
)  
RETURNS (  
    c INTEGER,
    str VARCHAR(100)
)  
AS  
BEGIN  
    -- assigning a constant  
    c = 0;
    str = '';  
    SUSPEND;
    -- assigning expression values  
    c = a + b;
    str = name || CAST(b AS VARCHAR(10));
    SUSPEND;
    -- assigning expression value  
    -- built by a query  
    c = (SELECT 1 FROM rdb$database);
    -- assigning a value from a context variable  
    str = CURRENT_USER;
    SUSPEND;
END
```

### See also

**DECLARE VARIABLE**

### 7.7.2. DECLARE VARIABLE

**Used for**
Declaring a local variable

**Available in**
PSQL

**Syntax**

```
DECLARE [VARIABLE] varname  
<domain_or_non_array_type> [NOT NULL] [COLLATE collation]  
[{{DEFAULT | = } <initvalue>}];
```
<domain_or_non_array_type> ::=  
!! See Scalar Data Types Syntax !!

<initvalue> ::= <literal> | <context_var>

Table 91. DECLARE VARIABLE Statement Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varname</td>
<td>Name of the local variable</td>
</tr>
<tr>
<td>collation</td>
<td>Collation sequence</td>
</tr>
<tr>
<td>initvalue</td>
<td>Initial value for this variable</td>
</tr>
<tr>
<td>literal</td>
<td>Literal of a type compatible with the type of the local variable</td>
</tr>
<tr>
<td>context_var</td>
<td>Any context variable whose type is compatible with the type of the local variable</td>
</tr>
</tbody>
</table>

The statement DECLARE [VARIABLE] is used for declaring a local variable. The keyword VARIABLE can be omitted. One DECLARE [VARIABLE] statement is required for each local variable. Any number of DECLARE [VARIABLE] statements can be included and in any order. The name of a local variable must be unique among the names of local variables and input and output parameters declared for the module.

A special case of DECLARE [VARIABLE]—declaring cursors—is covered separately in DECLARE .. CURSOR

Data Type for Variables

A local variable can be of any SQL type other than an array.

- A domain name can be specified as the type; the variable will inherit all of its attributes.
- If the TYPE OF domain clause is used instead, the variable will inherit only the domain's data type, and, if applicable, its character set and collation attributes. Any default value or constraints such as NOT NULL or CHECK constraints are not inherited.
- If the TYPE OF COLUMN relation.column> option is used to “borrow” from a column in a table or view, the variable will inherit only the column's data type, and, if applicable, its character set and collation attributes. Any other attributes are ignored.

NOT NULL Constraint

For local variables, you can specify the NOT NULL constraint, disallowing NULL values for the variable. If a domain has been specified as the data type and the domain already has the NOT NULL constraint, the declaration is unnecessary. For other forms, including use of a domain that is nullable, the NOT NULL constraint can be included if needed.

CHARACTER SET and COLLATE clauses

Unless specified, the character set and collation sequence of a string variable will be the database
defaults. A CHARACTER SET clause can be included, if required, to handle string data that is going to be in a different character set. A valid collation sequence (COLLATE clause) can also be included, with or without the character set clause.

**Initializing a Variable**

Local variables are NULL when execution of the module begins. They can be initialized so that a starting or default value is available when they are first referenced. The DEFAULT <initvalue> form can be used, or just the assignment operator, '=': = <initvalue>. The value can be any type-compatible literal or context variable, including NULL.

Be sure to use this clause for any variables that have a NOT NULL constraint and do not otherwise have a default value available.

**Examples of various ways to declare local variables**

```sql
CREATE OR ALTER PROCEDURE SOME_PROC
AS
  -- Declaring a variable of the INT type
  DECLARE I INT;
  -- Declaring a variable of the INT type that does not allow NULL
  DECLARE VARIABLE J INT NOT NULL;
  -- Declaring a variable of the INT type with the default value of 0
  DECLARE VARIABLE K INT DEFAULT 0;
  -- Declaring a variable of the INT type with the default value of 1
  DECLARE VARIABLE L INT = 1;
  -- Declaring a variable based on the COUNTRYNAME domain
  DECLARE FARM_COUNTRY COUNTRYNAME;
  -- Declaring a variable of the type equal to the COUNTRYNAME domain
  DECLARE FROM_COUNTRY TYPE OF COUNTRYNAME;
  -- Declaring a variable with the type of the CAPITAL column in the COUNTRY table
  DECLARE CAPITAL TYPE OF COLUMN COUNTRY.CAPITAL;
BEGIN
  /* PSQL statements */
END
```

See also

Data Types and Subtypes, Custom Data Types — Domains, CREATE DOMAIN

**7.7.3. DECLARE .. CURSOR**

*Used for*

Declaring a named cursor

*Available in*

PSQL
Syntax

DECLARE [VARIABLE] cursor_name
[[NO] SCROLL] CURSOR
FOR (<select>);

Table 92. DECLARE … CURSOR Statement Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursor_name</td>
<td>Cursor name</td>
</tr>
<tr>
<td>select</td>
<td>SELECT statement</td>
</tr>
</tbody>
</table>

The DECLARE … CURSOR … FOR statement binds a named cursor to the result set obtained in the SELECT statement specified in the FOR clause. In the body code, the cursor can be opened, used to iterate row-by-row through the result set, and closed. While the cursor is open, the code can perform positioned updates and deletes using the WHERE CURRENT OF in the UPDATE or DELETE statement.

Syntactically, the DECLARE … CURSOR statement is a special case of DECLARE VARIABLE.

Forward-Only and Scrollable Cursors

The cursor can be forward-only (unidirectional) or scrollable. The optional clause SCROLL makes the cursor scrollable, the NO SCROLL clause, forward-only. By default, cursors are forward-only.

Forward-only cursors can — as the name implies — only move forward in the dataset. Forward-only cursors only support the FETCH [NEXT FROM] statement, other commands raise an error. Scrollable cursors allow you to move not only forward in the dataset, but also back, as well as $N$ positions relative to the current position.

Scrollable cursors are materialized as a temporary dataset, as such, they consume additional memory or disk space, so use them only when you really need them.

Cursor Idiosyncrasies

- The optional FOR UPDATE clause can be included in the SELECT statement, but its absence does not prevent successful execution of a positioned update or delete
- Care should be taken to ensure that the names of declared cursors do not conflict with any names used subsequently in statements for AS CURSOR clauses
- If the cursor is needed only to walk the result set, it is nearly always easier and less error-prone to use a FOR SELECT statement with the AS CURSOR clause. Declared cursors must be explicitly opened, used to fetch data, and closed. The context variable ROW_COUNT has to be checked after each fetch and, if its value is zero, the loop has to be terminated. A FOR SELECT statement does this automatically.

Nevertheless, declared cursors provide a high level of control over sequential events and allow several cursors to be managed in parallel.
The SELECT statement may contain parameters. For instance:

```
SELECT NAME || :SFX FROM NAMES WHERE NUMBER = :NUM
```

Each parameter has to have been declared beforehand as a PSQL variable, even if they originate as input and output parameters. When the cursor is opened, the parameter is assigned the current value of the variable.

---

### Unstable Variables and Cursors

If the value of the PSQL variable used in the SELECT statement of the cursor changes during the execution of the loop, then its new value may—but not always—be used when selecting the next rows. It is better to avoid such situations. If you really need this behaviour, then you should thoroughly test your code and make sure you understand how changes to the variable affect the query results.

Note particularly that the behaviour may depend on the query plan, specifically on the indexes being used. Currently, there are no strict rules for this behaviour, and this may change in future versions of Firebird.

---

### Examples Using Named Cursors

1. Declaring a named cursor in the trigger.

   ```sql
   CREATE OR ALTER TRIGGER TBU_STOCK
   BEFORE UPDATE ON STOCK
   AS
   DECLARE C_COUNTRY CURSOR FOR (
       SELECT
           COUNTRY,
           CAPITAL
       FROM COUNTRY
   );
   BEGIN
   /* PSQL statements */
   END
   ```

2. Declaring a scrollable cursor

   ```sql
   EXECUTE BLOCK
   RETURNS ( 
       N INT,
       RNAME CHAR(31))
   AS
   - Declaring a scrollable cursor
   DECLARE C SCROLL CURSOR FOR ( 
       SELECT
   ```
EXECUTE BLOCK
RETURNS (  
  SCRIPT BLOB SUB_TYPE TEXT)
AS
  DECLARE VARIABLE FIELDS VARCHAR(8191);
  DECLARE VARIABLE FIELD_NAME TYPE OF RDB$FIELD_NAME;
  DECLARE VARIABLE RELATION RDB$RELATION_NAME;
  DECLARE VARIABLE SOURCE TYPE OF COLUMN RDB$RELATIONS.RDB$VIEW_SOURCE;
  DECLARE VARIABLE CUR_R CURSOR FOR (  
    SELECT  
      RDB$RELATION_NAME,
      RDB$VIEW_SOURCE
    FROM
      RDB$RELATIONS
    WHERE
      RDB$VIEW_SOURCE IS NOT NULL);
  -- Declaring a named cursor where
  -- a local variable is used
  DECLARE CUR_F CURSOR FOR (  
    SELECT  
      RDB$FIELD_NAME
    FROM
      RDB$RELATION_FIELDS
    WHERE
      -- It is important that the variable must be declared earlier
      RDB$RELATION_NAME = :RELATION);
BEGIN
  OPEN CUR_R;
  WHILE (1 = 1) DO
    BEGIN
      FETCH CUR_R
      INTO :RELATION, :SOURCE;
      IF (ROW_COUNT = 0) THEN
        LEAVE;
      END;
    END;
    FIELDS = NULL;
    -- The CUR_F cursor will use the value
    -- of the RELATION variable initiated above
  OPEN CUR_F;
  WHILE (1 = 1) DO
BEGIN
  FETCH CUR_F
  INTO :FIELD_NAME;
  IF (ROW_COUNT = 0) THEN
    LEAVE;
  END
  IF (FIELDS IS NULL) THEN
    FIELDS = TRIM(FIELD_NAME);
  ELSE
    FIELDS = FIELDS || ', ' || TRIM(FIELD_NAME);
  END
  CLOSE CUR_F;

  SCRIPT = 'CREATE VIEW ' || RELATION;
  IF (FIELDS IS NOT NULL) THEN
    SCRIPT = SCRIPT || ' (' || FIELDS || ');
  END
  SCRIPT = SCRIPT || ' AS ' || ASCII_CHAR(13);
  SCRIPT = SCRIPT || SOURCE;
  SUSPEND;
END
CLOSE CUR_R;
END

See also
OPEN, FETCH, CLOSE

7.7.4. DECLARE FUNCTION

Used for
Declaring a sub-function

Available in
PSQL

Syntax

```
DECLARE FUNCTION subfuncname [ ( [ <in_params> ] ) ]
  RETURNS <domain_or_non_array_type> [COLLATE collation]
  [DETERMINISTIC]
  <psql-module-body>
```

```
<in_params> ::=  
  !! See CREATE FUNCTION Syntax !!

<domain_or_non_array_type> ::=  
  !! See Scalar Data Types Syntax !!

<psql-module-body> ::=  
```
Table 93. DECLARE FUNCTION Statement Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subfuncname</td>
<td>Sub-function name</td>
</tr>
<tr>
<td>collation</td>
<td>Collation name</td>
</tr>
</tbody>
</table>

The DECLARE FUNCTION statement declares a sub-function. A sub-function is only visible to the PSQL module that defined the sub-function.

Sub-functions have a number of restrictions:

- A sub-function cannot be nested in another sub-routine. Sub-routines are only supported in top-level PSQL modules (stored procedures, stored functions, triggers and anonymous PSQL blocks). This restriction is not enforced by the syntax, but attempts to create nested sub-functions will raise an error “feature is not supported” with detail message “nested sub function”.

- Currently, the sub-function has no direct access to use variables, cursors and other routines (including itself) from its parent module. This may change in a future Firebird version.
  - As a result of this restriction, a sub-function cannot call itself recursively; attempts to call it will yield error “Function unknown: subfuncname”.

- Declaring a sub-function with the same name as a stored function will hide that stored function from your module. It will not be possible to call that stored function.

- Contrary to DECLARE [VARIABLE], a DECLARE FUNCTION is not terminated by a semicolon. The END of its main BEGIN … END block is considered its terminator.

Examples of Sub-Functions

Subfunction within a stored function

```sql
CREATE OR ALTER FUNCTION FUNC1 (n1 INTEGER, n2 INTEGER)
RETURNS INTEGER
AS
- Subfunction
  DECLARE FUNCTION SUBFUNC (n1 INTEGER, n2 INTEGER)
  RETURNS INTEGER
  AS
  BEGIN
  RETURN n1 + n2;
  END
BEGIN
RETURN SUBFUNC (n1, n2);
END
```
7.7.5. DECLARE PROCEDURE

Used for
Declaring a sub-procedure

Available in
PSQL

Syntax

```
DECLARE subprocname [ ( [ <in_params> ] ) ]
[RETURNS (<out_params>)]
<psq-module-body>
```

- `<in_params>` ::= !! See CREATE PROCEDURE Syntax !!
- `<domain_or_non_array_type>` ::= !! See Scalar Data Types Syntax !!
- `<psql-module-body>` ::= !! See Syntax of Module Body !!

**Table 94. DECLARE PROCEDURE Statement Parameters**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subprocname</td>
<td>Sub-procedure name</td>
</tr>
<tr>
<td>collation</td>
<td>Collation name</td>
</tr>
</tbody>
</table>

The DECLARE PROCEDURE statement declares a sub-procedure. A sub-procedure is only visible to the PSQL module that defined the sub-procedure.

Sub-procedures have a number of restrictions:

- A sub-procedure cannot be nested in another sub-routine. Sub-routines are only supported in top-level PSQL modules (stored procedures, stored functions, triggers and anonymous PSQL blocks). This restriction is not enforced by the syntax, but attempts to create nested sub-procedures will raise an error “feature is not supported” with detail message “nested sub-procedure”.

- Currently, the sub-procedure has no direct access to use variables, cursors and other routines (including itself) from its parent module. This may change in a future Firebird version.
  - As a result of this restriction, a sub-procedure cannot call itself recursively; attempts to call it will yield error “Function unknown: subprocname”.

  Declaring a sub-procedure with the same name as a stored procedure, table or
view will hide that stored procedure, table or view from your module. It will not be possible to call that stored procedure, table or view.

Contrary to DECLARE [VARIABLE], a DECLARE PROCEDURE is not terminated by a semicolon. The END of its main BEGIN … END block is considered its terminator.

Examples of Sub-Procedures

Subroutines in EXECUTE BLOCK

EXECUTE BLOCK
   RETURNS (name VARCHAR(31))
AS
  -- Sub-procedure returning a list of tables
  DECLARE PROCEDURE get_tables
     RETURNS (table_name VARCHAR(31))
     AS
     BEGIN
        FOR SELECT RDB$RELATION_NAME
           FROM RDB$RELATIONS
           WHERE RDB$VIEW_BLR IS NULL
           INTO table_name
        DO SUSPEND;
     END
  -- Sub-procedure returning a list of views
  DECLARE PROCEDURE get_views
     RETURNS (view_name VARCHAR(31))
     AS
     BEGIN
        FOR SELECT RDB$RELATION_NAME
           FROM RDB$RELATIONS
           WHERE RDB$VIEW_BLR IS NOT NULL
           INTO view_name
        DO SUSPEND;
     END
  BEGIN
     FOR SELECT table_name
        FROM get_tables
     UNION ALL
     SELECT view_name
        FROM get_views
     INTO name
     DO SUSPEND;
  END

See also

DECLARE FUNCTION, CREATE PROCEDURE
7.7.6. **BEGIN … END**

**Used for**
Delimiting a block of statements

**Available in**
PSQL

**Syntax**

```plaintext
<block> ::= 
BEGIN
   [<compound_statement> ...]
END

<compound_statement> ::= {<block> | <statement>}
```

The `BEGIN … END` construct is a two-part statement that wraps a block of statements that are executed as one unit of code. Each block starts with the half-statement `BEGIN` and ends with the other half-statement `END`. Blocks can be nested a maximum depth of 512 nested blocks. A block can be empty, allowing them to act as stubs, without the need to write dummy statements.

The `BEGIN` and `END` statements have no line terminators (semicolon). However, when defining or altering a PSQL module in the `isql` utility, that application requires that the last `END` statement be followed by its own terminator character, that was previously switched — using `SET TERM` — to some string other than a semicolon. That terminator is not part of the PSQL syntax.

The final, or outermost, `END` statement in a trigger terminates the trigger. What the final `END` statement does in a stored procedure depends on the type of procedure:

- In a selectable procedure, the final `END` statement returns control to the caller, returning SQLCODE 100, indicating that there are no more rows to retrieve
- In an executable procedure, the final `END` statement returns control to the caller, along with the current values of any output parameters defined.

**BEGIN … END Examples**

A sample procedure from the `employee.fdb` database, showing simple usage of `BEGIN…END` blocks:

```sql
SET TERM ^;
CREATE OR ALTER PROCEDURE DEPT_BUDGET ( 
   DNO CHAR(3))
RETURNS ( 
   TOT DECIMAL(12,2))
AS
   DECLARE VARIABLE SUMB DECIMAL(12,2);
   DECLARE VARIABLE RDNO CHAR(3);
   DECLARE VARIABLE CNT  INTEGER;
BEGIN
```
TOT = 0;

SELECT BUDGET
FROM DEPARTMENT
WHERE DEPT_NO = :DNO
INTO :TOT;

SELECT COUNT(BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = :DNO
INTO :CNT;

IF (CNT = 0) THEN
    SUSPEND;

FOR SELECT DEPT_NO
    FROM DEPARTMENT
    WHERE HEAD_DEPT = :DNO
    INTO :RDNO
DO
BEGIN
    EXECUTE PROCEDURE DEPT_BUDGET(:RDNO)
    RETURNING_VALUES :SUMB;
    TOT = TOT + SUMB;
END

    SUSPEND;
END^;
SET TERM ;^;

See also
EXIT, SET TERM

7.7.7. IF … THEN … ELSE

Used for
Conditional branching

Available in
PSQL

Syntax

IF (<condition>)
    THEN <compound_statement>
    [ELSE <compound_statement>]

Table 95. IF … THEN … ELSE Parameters
The conditional branch statement **IF … THEN** is used to branch the execution process in a PSQL module. The condition is always enclosed in parentheses. If the condition returns the value TRUE, execution branches to the statement or the block of statements after the keyword **THEN**. If an **ELSE** is present, and the condition returns FALSE or UNKNOWN, execution branches to the statement or the block of statements after it.

### Multi-Branch Decisions

PSQL does not provide more advanced multi-branch jumps, such as **CASE** or **SWITCH**. However, it is possible to chain **IF … THEN … ELSE** statements, see the example section below. Alternatively, the **CASE** statement from DSQL is available in PSQL and is able to satisfy at least some use cases in the manner of a switch:

```sql
CASE <test_expr>
    WHEN <expr> THEN <result>
    [WHEN <expr> THEN <result> ...]
    [ELSE <defaultresult>]
END

CASE
    WHEN <bool_expr> THEN <result>
    [WHEN <bool_expr> THEN <result> ...]
    [ELSE <defaultresult>]
END
```

**Example in PSQL**

```sql
... 
C = CASE
    WHEN A=2 THEN 1
    WHEN A=1 THEN 3
    ELSE 0
END;
... 
```

### If Examples

1. An example using the **If** statement. Assume that the **FIRST**, **LINE2** and **LAST** variables were declared earlier.
IF (FIRST IS NOT NULL) THEN
    LINE2 = FIRST || ' ' || LAST;
ELSE
    LINE2 = LAST;
...

2. Given IF ... THEN ... ELSE is a statement, it is possible to chain them together. Assume that the INT_VALUE and STRING_VALUE variables were declared earlier.

IF (INT_VALUE = 1) THEN
    STRING_VALUE = 'one';
ELSE IF (INT_VALUE = 2) THEN
    STRING_VALUE = 'two';
ELSE IF (INT_VALUE = 3) THEN
    STRING_VALUE = 'three';
ELSE
    STRING_VALUE = 'too much';

This specific example can be replaced with a simple CASE or the DECODE function.

See also
WHILE ... DO, CASE

7.7.8. WHILE ... DO

Used for
Looping constructs

Available in
PSQL

Syntax

[<label>]
WHILE <condition> DO
    <compound_statement>

Table 96. WHILE ... DO Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>Optional label for LEAVE and CONTINUE. Follows the rules for identifiers.</td>
</tr>
<tr>
<td>condition</td>
<td>A logical condition returning TRUE, FALSE or UNKNOWN</td>
</tr>
<tr>
<td>compound_statement</td>
<td>A single statement, or two or more statements wrapped in BEGIN ... END</td>
</tr>
</tbody>
</table>

A WHILE statement implements the looping construct in PSQL. The statement or the block of statements will be executed until the condition returns TRUE. Loops can be nested to any depth.
WHILE … DO Examples

A procedure calculating the sum of numbers from 1 to I shows how the looping construct is used.

CREATE PROCEDURE SUM_INT (I INTEGER)
RETURNS (S INTEGER)
AS
BEGIN
    s = 0;
    WHILE (i > 0) DO
        BEGIN
            s = s + i;
            i = i - 1;
        END
    END
END

Executing the procedure in *isql*:

EXECUTE PROCEDURE SUM_INT(4);

the result is:

\[
\begin{array}{l}
S \\
\hline
\text{========} \\
10 \\
\end{array}
\]

See also

IF … THEN … ELSE, BREAK, LEAVE, CONTINUE, EXIT, FOR SELECT, FOR EXECUTE STATEMENT

7.7.9. BREAK

*Used for*

Exiting a loop

*Available in*

PSQL

*Syntax*

\[
[label:] \\
<loop_stmt> \\
BEGIN \\
... \\
BREAK; \\
... \\
END
\]
The `BREAK` statement immediately terminates the inner loop of a `WHILE` or `FOR` looping statement. Code continues to be executed from the first statement after the terminated loop block.

`BREAK` is similar to `LEAVE`, except it doesn’t support a label.

**See also**

`LEAVE`

### 7.7.10. LEAVE

**Used for**

Exiting a loop

**Available in**

PSQL

**Syntax**

```
[label:]
<loop_stmt>
BEGIN
...
  LEAVE [label];
...
END
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>Label</td>
</tr>
<tr>
<td>select_stmt</td>
<td>SELECT statement</td>
</tr>
<tr>
<td>condition</td>
<td>A logical condition returning TRUE, FALSE or UNKNOWN</td>
</tr>
</tbody>
</table>

**Table 98. LEAVE Statement Parameters**
### Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>A logical condition returning TRUE, FALSE or UNKNOWN</td>
</tr>
</tbody>
</table>

The **LEAVE** statement immediately terminates the inner loop of a **WHILE** or **FOR** looping statement. Using the optional *label* parameter, **LEAVE** can also exit an outer loop, that is, the loop labelled with *label*. Code continues to be executed from the first statement after the terminated loop block.

**LEAVE Examples**

1. Leaving a loop if an error occurs on an insert into the **NUMBERS** table. The code continues to be executed from the line `C = 0`.

   ```sql
   ... 
   WHILE (B < 10) DO 
   BEGIN 
   INSERT INTO NUMBERS(B) 
   VALUES (:B); 
   B = B + 1; 
   WHEN ANY DO 
   BEGIN 
   EXECUTE PROCEDURE LOG_ERROR ( 
   CURRENT_TIMESTAMP, 
   'ERROR IN B LOOP'); 
   LEAVE; 
   END 
   END 
   C = 0; 
   ... 
   ```

2. An example using labels in the **LEAVE** statement. **LEAVE LOOPA** terminates the outer loop and **LEAVE LOOPB** terminates the inner loop. Note that the plain **LEAVE** statement would be enough to terminate the inner loop.

   ```sql
   ... 
   STMT1 = 'SELECT NAME FROM FARMS'; 
   LOOPA: 
   FOR EXECUTE STATEMENT :STMT1 
   INTO :FARM DO 
   BEGIN 
   STMT2 = 'SELECT NAME ' || 'FROM ANIMALS WHERE FARM = '''; 
   LOOPB: 
   FOR EXECUTE STATEMENT :STMT2 || :FARM || '''
   INTO :ANIMAL DO 
   BEGIN 
   IF (ANIMAL = 'FLUFFY') THEN 
   LEAVE LOOPB; 
   ELSE IF (ANIMAL = FARM) THEN 
   LEAVE LOOPA; 
   ```
See also
BREAK, CONTINUE, EXIT

7.7.11. CONTINUE

Used for
Continuing with the next iteration of a loop

Available in
PSQL

Syntax

```
[label:] <loop_stmt> BEGIN ...
  CONTINUE [label];
...
END

<loop_stmt> ::= 
  FOR <select_stmt> INTO <var_list> DO 
  | FOR EXECUTE STATEMENT ... INTO <var_list> DO 
  | WHILE (<condition>)} DO
```

Table 99. CONTINUE Statement Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>Label</td>
</tr>
<tr>
<td>select_stmt</td>
<td>SELECT statement</td>
</tr>
<tr>
<td>condition</td>
<td>A logical condition returning TRUE, FALSE or UNKNOWN</td>
</tr>
</tbody>
</table>

The CONTINUE statement skips the remainder of the current block of a loop and starts the next iteration of the current WHILE or FOR loop. Using the optional label parameter, CONTINUE can also start the next iteration of an outer loop, that is, the loop labelled with label.

CONTINUE Examples

Using the CONTINUE statement

```
FOR SELECT A, D
```
FROM ATABLE INTO achar, ddate
DO
BEGIN
  IF (ddate < current_date - 30) THEN
    CONTINUE;
  ELSE
    BEGIN
      /* do stuff */
    END
  END
END

See also
BREAK, LEAVE, EXIT

7.7.12. EXIT

Used for
Terminating module execution

Available in
PSQL

Syntax

EXIT;

The EXIT statement causes execution of the current PSQL module to jump to the final END statement from any point in the code, thus terminating the program.

Calling EXIT in a function will result in the function returning NULL.

EXIT Examples

Using the EXIT statement in a selectable procedure

CREATE PROCEDURE GEN_100
  RETURNS (I INTEGER)
AS
BEGIN
  I = 1;
  WHILE (I=1) DO
    BEGIN
      SUSPEND;
      IF (I=100) THEN
        EXIT;
      I = I + 1;
    END
  END
7.7.13. SUSPEND

Used for
Passing output to the buffer and suspending execution while waiting for caller to fetch it

Available in
PSQL

Syntax

SUSPEND;

The SUSPEND statement is used in a selectable stored procedure to pass the values of output parameters to a buffer and suspend execution. Execution remains suspended until the calling application fetches the contents of the buffer. Execution resumes from the statement directly after the SUSPEND statement. In practice, this is likely to be a new iteration of a looping process.

Important Notes

1. The SUSPEND statement can only occur in stored procedures or sub-procedures
2. The presence of the SUSPEND keyword defines a stored procedure as a selectable procedure
3. Applications using interfaces that wrap the API perform the fetches from selectable procedures transparently.
4. If a selectable procedure is executed using EXECUTE PROCEDURE, it behaves as an executable procedure. When a SUSPEND statement is executed in such a stored procedure, it is the same as executing the EXIT statement, resulting in immediate termination of the procedure.
5. SUSPEND “breaks” the atomicity of the block in which it is located. If an error occurs in a selectable procedure, statements executed after the final SUSPEND statement will be rolled back. Statements that executed before the final SUSPEND statement will not be rolled back unless the transaction is rolled back.

SUSPEND Examples

Using the SUSPEND statement in a selectable procedure

```
CREATE PROCEDURE GEN_100
  RETURNS (I INTEGER)
AS
BEGIN
  I = 1;
  WHILE (1=1) DO
    SUSPEND;
    I = I + 1;
  END;
END;
```
SUSPEND;
IF (I=100) THEN
EXIT;
I = I + 1;
END
END

See also
EXIT

7.7.14. EXECUTE STATEMENT

Used for
Executing dynamically created SQL statements

Available in
PSQL

Syntax

<execute_statement> ::= EXECUTE STATEMENT <argument>
[<option> ...]
[INTO <variables>];

<argument> ::= <paramless_stmt>
| (<paramless_stmt>)
| (stmt_with_params) (<param_values>)

<param_values> ::= <named_values> | <positional_values>

<named_values> ::= paramname := <value_expr>
[,, paramname := <value_expr> ...]

<positional_values> ::= <value_expr> [, <value_expr> ...]

<option> ::= WITH {AUTONOMOUS | COMMON} TRANSACTION
| WITH CALLER PRIVILEGES
| AS USER user
| PASSWORD password
| ROLE role
| ON EXTERNAL [DATA SOURCE] <connection_string>

<connection_string> ::= !! See <filespec> in the CREATE DATABASE syntax !!

<variables> ::= [:]varname [, [:]varname ...]

Table 100. EXECUTE STATEMENT Statement Parameters
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramless_stmt</td>
<td>Literal string or variable containing a non-parameterized SQL query</td>
</tr>
<tr>
<td>stmt_with_params</td>
<td>Literal string or variable containing a parameterized SQL query</td>
</tr>
<tr>
<td>paramname</td>
<td>SQL query parameter name</td>
</tr>
<tr>
<td>value_expr</td>
<td>SQL expression resolving to a value</td>
</tr>
<tr>
<td>user</td>
<td>Username. It can be a string, CURRENT_USER or a string variable</td>
</tr>
<tr>
<td>password</td>
<td>Password. It can be a string or a string variable</td>
</tr>
<tr>
<td>role</td>
<td>Role. It can be a string, CURRENT_ROLE or a string variable</td>
</tr>
<tr>
<td>connection_string</td>
<td>Connection string. It can be a string literal or a string variable</td>
</tr>
<tr>
<td>varname</td>
<td>Variable</td>
</tr>
</tbody>
</table>

The statement **EXECUTE STATEMENT** takes a string parameter and executes it as if it were a DSQL statement. If the statement returns data, it can be passed to local variables by way of an INTO clause.

**EXECUTE STATEMENT** can only produce a single row of data. Statements producing multiple rows of data must be executed with **FOR EXECUTE STATEMENT**.

**Parameterized Statements**

You can use parameters—either named or positional—in the DSQL statement string. Each parameter must be assigned a value.

**Special Rules for Parameterized Statements**

1. Named and positional parameters cannot be mixed in one query
2. If the statement has parameters, they must be enclosed in parentheses when **EXECUTE STATEMENT** is called, regardless of whether they come directly as strings, as variable names or as expressions
3. Each named parameter must be prefixed by a colon (‘:’) in the statement string itself, but not when the parameter is assigned a value
4. Positional parameters must be assigned their values in the same order as they appear in the query text
5. The assignment operator for parameters is the special operator “:=”, similar to the assignment operator in Pascal
6. Each named parameter can be used in the statement more than once, but its value must be assigned only once
7. With positional parameters, the number of assigned values must match the number of parameter placeholders (question marks) in the statement exactly
8. A named parameter in the statement text can only be a regular identifier (it cannot be a quoted identifier)
Examples of `EXECUTE STATEMENT` with parameters

With named parameters:

```
DECLARE license_num VARCHAR(15);
DECLARE connect_string VARCHAR(100);
DECLARE stmt VARCHAR(100) =
    'SELECT license
     FROM cars
     WHERE driver = :driver AND location = :loc';
BEGIN
    ...
    SELECT connstr
    FROM databases
    WHERE cust_id = :id
    INTO connect_string;
    ...
    FOR
    SELECT id
    FROM drivers
    INTO current_driver
    DO
    BEGIN
    FOR
    SELECT location
    FROM driver_locations
    WHERE driver_id = :current_driver
    INTO current_location
    DO
    BEGIN
    ...
    EXECUTE STATEMENT (stmt)
        (driver := current_driver,
        loc := current_location)
    ON EXTERNAL connect_string
    INTO license_num;
    ...
```

The same code with positional parameters:

```
DECLARE license_num VARCHAR(15);
DECLARE connect_string VARCHAR(100);
DECLARE stmt VARCHAR(100) =
    'SELECT license
     FROM cars
     WHERE driver = ? AND location = ?';
BEGIN
    ...
    SELECT connstr
```
FOR SELECT id FROM drivers INTO current_driver DO BEGIN FOR SELECT location FROM driver_locations WHERE driver_id = :current_driver INTO current_location DO BEGIN ... EXECUTE STATEMENT (stmt) (current_driver, current_location) ON EXTERNAL connect_string INTO license_num; ...

**WITH {AUTONOMOUS | COMMON} TRANSACTION**

By default, the executed SQL statement runs within the current transaction. Using **WITH AUTONOMOUS TRANSACTION** causes a separate transaction to be started, with the same parameters as the current transaction. This separate transaction will be committed when the statement was executed without errors and rolled back otherwise.

The clause **WITH COMMON TRANSACTION** uses the current transaction whenever possible; this is the default behaviour. If the statement must run in a separate connection, an already started transaction within that connection is used, if available. Otherwise, a new transaction is started with the same parameters as the current transaction. Any new transactions started under the “COMMON” regime are committed or rolled back with the current transaction.

**WITH CALLER PRIVILEGES**

By default, the SQL statement is executed with the privileges of the current user. Specifying **WITH CALLER PRIVILEGES** combines the privileges of the calling procedure or trigger with those of the user, just as if the statement were executed directly by the routine. **WITH CALLER PRIVILEGES** has no effect if the **ON EXTERNAL** clause is also present.

**ON EXTERNAL [DATA SOURCE]**

With **ON EXTERNAL [DATA SOURCE]**, the SQL statement is executed in a separate connection to the same or another database, possibly even on another server. If **connection_string** is NULL or “” (empty string), the entire **ON EXTERNAL [DATA SOURCE]** clause is considered absent, and the statement is executed against the current database.
Connection Pooling

- External connections made by statements WITH COMMON TRANSACTION (the default) will remain open until the current transaction ends. They can be reused by subsequent calls to EXECUTE STATEMENT, but only if connection_string is exactly the same, including case.

- External connections made by statements WITH AUTONOMOUS TRANSACTION are closed as soon as the statement has been executed.

- Statements using WITH AUTONOMOUS TRANSACTION can and will re-use connections that were opened earlier by statements WITH COMMON TRANSACTION. If this happens, the reused connection will be left open after the statement has been executed. (It must be, because it has at least one active transaction!)

Transaction Pooling

- If WITH COMMON TRANSACTION is in effect, transactions will be reused as much as possible. They will be committed or rolled back together with the current transaction.

- If WITH AUTONOMOUS TRANSACTION is specified, a fresh transaction will always be started for the statement. This transaction will be committed or rolled back immediately after the statement's execution.

Exception Handling

When ON EXTERNAL is used, the extra connection is always made via a so-called external provider, even if the connection is to the current database. One of the consequences is that exceptions cannot be caught in the usual way. Every exception caused by the statement is wrapped in either an eds_connection or an eds_statement error. In order to catch them in your PSQL code, you have to use WHEN GDSCODE eds_connection, WHEN GDSCODE eds_statement or WHEN ANY.

Without ON EXTERNAL, exceptions are caught in the usual way, even if an extra connection is made to the current database.

Miscellaneous Notes

- The character set used for the external connection is the same as that for the current connection.

- Two-phase commits are not supported.

AS USER, PASSWORD and ROLE

The optional AS USER, PASSWORD and ROLE clauses allow specification of which user will execute the SQL statement and with which role. The method of user login, and whether a separate connection is opened, depends on the presence and values of the ON EXTERNAL [DATA SOURCE], AS USER, PASSWORD and ROLE clauses:

- If ON EXTERNAL is present, a new connection is always opened, and:
  - If at least one of AS USER, PASSWORD and ROLE is present, native authentication is attempted with the given parameter values (locally or remotely, depending on connection_string). No defaults are used for missing parameters.
  - If all three are absent, and connection_string contains no hostname, then the new
connection is established on the local server with the same user and role as the current connection. The term 'local' means “on the same machine as the server” here. This is not necessarily the location of the client

- If all three are absent, and connection_string contains a hostname, then trusted authentication is attempted on the remote host (again, 'remote' from the perspective of the server). If this succeeds, the remote operating system will provide the username (usually the operating system account under which the Firebird process runs)

- If ON EXTERNAL is absent:
  - If at least one of AS USER, PASSWORD and ROLE is present, a new connection to the current database is opened with the supplied parameter values. No defaults are used for missing parameters
  - If all three are absent, the statement is executed within the current connection

If a parameter value is NULL or “'” (empty string), the entire parameter is considered absent. Additionally, AS USER is considered absent if its value is equal to CURRENT_USER, and ROLE if it is the same as CURRENT_ROLE.

Caveats with EXECUTE STATEMENT

1. There is no way to validate the syntax of the enclosed statement
2. There are no dependency checks to discover whether tables or columns have been dropped
3. Even though the performance in loops has been significantly improved in Firebird 2.5, execution is still considerably slower than when the same statements are executed directly
4. Return values are strictly checked for data type in order to avoid unpredictable type-casting exceptions. For example, the string '1234' would convert to an integer, 1234, but 'abc' would give a conversion error

All in all, this feature is meant to be used very cautiously, and you should always take the caveats into account. If you can achieve the same result with PSQL and/or DSQL, it will almost always be preferable.

See also
FOR EXECUTE STATEMENT

7.7.15. FOR SELECT

Used for
Looping row-by-row through a selected result set

Available in
PSQL

Syntax

```
[setLabel:] FOR <select_stmt> [AS CURSOR cursor_name]
```
DO <compound_statement>

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>Optional label for LEAVE and CONTINUE. Follows the rules for identifiers.</td>
</tr>
<tr>
<td>select_stmt</td>
<td>SELECT statement</td>
</tr>
<tr>
<td>cursor_name</td>
<td>Cursor name. It must be unique among cursor names in the PSQL module (stored procedure, stored function, trigger or PSQL block)</td>
</tr>
<tr>
<td>compound_statement</td>
<td>A single statement, or a block of statements wrapped in BEGIN…END, that performs all the processing for this FOR loop</td>
</tr>
</tbody>
</table>

The **FOR SELECT** statement

- retrieves each row sequentially from the result set, and executes the statement or block of statements for each row. In each iteration of the loop, the field values of the current row are copied into pre-declared variables.

  Including the `AS CURSOR` clause enables positioned deletes and updates to be performed — see notes below

- can embed other **FOR SELECT** statements
- can contain named parameters that must be previously declared in the `DECLARE VARIABLE` statement or exist as input or output parameters of the procedure
- requires an `INTO` clause at the end of the `SELECT ... FROM ...` specification. In each iteration of the loop, the field values of the current row are copied to the list of variables specified in the `INTO` clause. The loop repeats until all rows are retrieved, after which it terminates
- can be terminated before all rows are retrieved by using a `BREAK`, `LEAVE` or `EXIT` statement

### The Undeclared Cursor

The optional `AS CURSOR` clause surfaces the set in the FOR SELECT structure as an undeclared, named cursor that can be operated on using the `WHERE CURRENT OF` clause inside the statement or block following the `DO` command, in order to delete or update the current row before execution moves to the next row. In addition, it is possible to use the cursor name as a record variable (similar to `OLD` and `NEW` in triggers), allowing access to the columns of the result set (i.e. `cursor_name.columnname`).

### Rules for Cursor Variables

- When accessing a cursor variable in a DML statement, the colon prefix can be added before the `cursor name` (i.e. `:cursor_name.columnname`) for disambiguation, similar to variables.

  The cursor variable can be referenced without colon prefix, but in that case, depending on the scope of the contexts in the statement, the name may resolve in the statement context instead of to the cursor (e.g. you select from a table with the same name as the cursor).

- Cursor variables are read-only
In a FOR SELECT statement without an AS CURSOR clause, you must use the INTO clause. If an AS CURSOR clause is specified, the INTO clause is allowed, but optional; you can access the fields through the cursor instead.

Reading from a cursor variable returns the current field values. This means that an UPDATE statement (with a WHERE CURRENT OF clause) will update not only the table, but also the fields in the cursor variable for subsequent reads. Executing a DELETE statement (with a WHERE CURRENT OF clause) will set all fields in the cursor variable to NULL for subsequent reads.

Other points to take into account regarding undeclared cursors:

1. The OPEN, FETCH and CLOSE statements cannot be applied to a cursor surfaced by the AS CURSOR clause.
2. The cursor_name argument associated with an AS CURSOR clause must not clash with any names created by DECLARE VARIABLE or DECLARE CURSOR statements at the top of the module body, nor with any other cursors surfaced by an AS CURSOR clause.
3. The optional FOR UPDATE clause in the SELECT statement is not required for a positioned update.

**Examples using FOR SELECT**

1. A simple loop through query results:

   ```sql
   CREATE PROCEDURE SHOWNUMS
   RETURNS ( 
       AA INTEGER,
       BB INTEGER,
       SM INTEGER,
       DF INTEGER)
   AS
   BEGIN
   FOR SELECT DISTINCT A, B 
   FROM NUMBERS
   ORDER BY A, B
   INTO AA, BB
   DO
   BEGIN
   SM = AA + BB;
   DF = AA - BB;
   SUSPEND;
   END
   END
   ```

2. Nested FOR SELECT loop:

   ```sql
   CREATE PROCEDURE RELFIELDS
   RETURNS ( 
       RELATION CHAR(32),
       POS INTEGER,
   ```
3. Using the AS CURSOR clause to surface a cursor for the positioned delete of a record:

```sql
CREATE PROCEDURE DELTOWN (  
    TOWNTODELETE VARCHAR(24))  
RETURNS (  
    TOWN VARCHAR(24),  
    POP INTEGER)  
AS  
BEGIN  
    FOR SELECT TOWN, POP  
        FROM TOWNS  
        INTO :TOWN, :POP AS CURSOR TCUR  
    DO  
        BEGIN  
            IF (:TOWN = :TOWNTODELETE) THEN  
                -- Positional delete  
                DELETE FROM TOWNS  
                WHERE CURRENT OF TCUR;  
            ELSE  
                SUSPEND;  
            END  
        END  
    END  
END
```
4. Using an implicitly declared cursor as a cursor variable

```sql
EXECUTE BLOCK
  RETURNS (o CHAR(31))
AS
BEGIN
  FOR SELECT rdb$relation_name AS name
    FROM rdb$relations AS CURSOR c
  DO
    BEGIN
      o = c.name;
      SUSPEND;
    END
  END
END
```

5. Disambiguating cursor variables within queries

```sql
EXECUTE BLOCK
  RETURNS (o1 CHAR(31), o2 CHAR(31))
AS
BEGIN
  FOR SELECT rdb$relation_name
    FROM rdb$relations
     WHERE
     rdb$relation_name = 'RDB$RELATIONS' AS CURSOR c
  DO
    BEGIN
      FOR SELECT
        -- with a prefix resolves as a cursor
        :c.rdb$relation_name x1,
        -- no prefix as an alias for the rdb$relations table
        c.rdb$relation_name x2
      FROM rdb$relations c
     WHERE
     rdb$relation_name = 'RDB$DATABASE' AS CURSOR d
    DO
      BEGIN
        o1 = d.x1;
        o2 = d.x2;
        SUSPEND;
      END
    END
  END
END
```

See also

`DECLARE .. CURSOR, BREAK, LEAVE, CONTINUE, EXIT, SELECT, UPDATE, DELETE`
**7.7.16. FOR EXECUTE STATEMENT**

*Used for*

Executing dynamically created SQL statements that return a row set

*Available in*

PSQL

*Syntax*

```
[label:]
FOR <execute_statement> DO <compound_statement>
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>Optional label for LEAVE and CONTINUE. Follows the rules for identifiers.</td>
</tr>
<tr>
<td>execute_stmt</td>
<td>An EXECUTE STATEMENT statement</td>
</tr>
<tr>
<td>compound_statement</td>
<td>A single statement, or a block of statements wrapped in BEGIN…END, that</td>
</tr>
<tr>
<td></td>
<td>performs all the processing for this FOR loop</td>
</tr>
</tbody>
</table>

The statement FOR EXECUTE STATEMENT is used, in a manner analogous to FOR SELECT, to loop through the result set of a dynamically executed query that returns multiple rows.

**FOR EXECUTE STATEMENT Examples**

*Executing a dynamically constructed SELECT query that returns a data set*

```sql
CREATE PROCEDURE DynamicSampleThree (
    Q_FIELD_NAME VARCHAR(100),
    Q_TABLE_NAME VARCHAR(100)
) RETURNS(
    LINE VARCHAR(32000)
) AS
    DECLARE VARIABLE P_ONE_LINE VARCHAR(100);
BEGIN
    LINE = '';
    FOR EXECUTE STATEMENT
        'SELECT T1.' || :Q_FIELD_NAME || ' FROM ' || :Q_TABLE_NAME || ' T1'
    INTO :P_ONE_LINE
    DO
        IF (:P_ONE_LINE IS NOT NULL) THEN
            LINE = :LINE || :P_ONE_LINE || ' ';
            SUSPEND;
        END
    END
```
See also
EXECUTE STATEMENT, BREAK, LEAVE, CONTINUE

7.7.17. OPEN

Used for
Opening a declared cursor

Available in
PSQL

Syntax

```
OPEN cursor_name;
```

Table 103. OPEN Statement Parameter

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursor_name</td>
<td>Cursor name. A cursor with this name must be previously declared with a DECLARE CURSOR statement</td>
</tr>
</tbody>
</table>

An OPEN statement opens a previously declared cursor, executes its declared SELECT statement, and makes the first record of the result data set ready to fetch. OPEN can be applied only to cursors previously declared in a DECLARE CURSOR statement.

If the SELECT statement of the cursor has parameters, they must be declared as local variables or exist as input or output parameters before the cursor is declared. When the cursor is opened, the parameter is assigned the current value of the variable.

OPEN Examples

1. Using the OPEN statement:

```
SET TERM ^;

CREATE OR ALTER PROCEDURE GET_RELATIONS_NAMES
RETURNS ( RNAME CHAR(31) )
AS
  DECLARE C CURSOR FOR (
    SELECT RDB$RELATION_NAME FROM RDB$RELATIONS);
BEGIN
  OPEN C;
  WHILE (1 = 1) DO
    BEGIN
```
2. A collection of scripts for creating views using a PSQL block with named cursors:

```sql
EXECUTE BLOCK
RETURNS (SCRIPT BLOB SUB_TYPE TEXT)
AS
    DECLARE VARIABLE FIELDS VARCHAR(8191);
    DECLARE VARIABLE FIELD_NAME TYPE OF RDB$FIELD_NAME;
    DECLARE VARIABLE RELATION RDB$RELATION_NAME;
    DECLARE VARIABLE SOURCE TYPE OF COLUMN RDB$RELATIONS.RDB$VIEW_SOURCE;
    -- named cursor
    DECLARE VARIABLE CUR_R CURSOR FOR (SELECT RDB$RELATION_NAME, RDB$VIEW_SOURCE FROM RDB$RELATIONS WHERE RDB$VIEW_SOURCE IS NOT NULL);
    -- named cursor with local variable
    DECLARE CUR_F CURSOR FOR (SELECT RDB$FIELD_NAME FROM RDB$RELATION_FIELDS WHERE -- Important! The variable has to be declared earlier RDB$RELATION_NAME = :RELATION);
BEGIN
    OPEN CUR_R;
    WHILE (1 = 1) DO
        BEGIN
            FETCH CUR_R INTO :RELATION, :SOURCE;
            IF (ROW_COUNT = 0) THEN
                LEAVE;
            END
            FIELDS = NULL;
            -- The CUR_F cursor will use
            -- variable value of RELATION initialized above
        END
    END
    CLOSE CUR_R;
END
```
OPEN CUR_F;
WHILE (1 = 1) DO 
BEGIN
  FETCH CUR_F
  INTO :FIELD_NAME;
  IF (ROW_COUNT = 0) THEN 
    LEAVE;
  IF (FIELDS IS NULL) THEN
    FIELDS = TRIM(FIELD_NAME);
  ELSE 
    FIELDS = FIELDS || ', ' || TRIM(FIELD_NAME);
  END 
CLOSE CUR_F;

SCRIPT = 'CREATE VIEW ' || RELATION;

IF (FIELDS IS NOT NULL) THEN 
  SCRIPT = SCRIPT || ' (' || FIELDS || ')';

SCRIPT = SCRIPT || ' AS ' || ASCII_CHAR(13);
SCRIPT = SCRIPT || SOURCE;

SUSPEND;
END
CLOSE CUR_R;
END

See also
DECLARE .. CURSOR, FETCH, CLOSE

7.7.18. FETCH

Used for
Fetching successive records from a data set retrieved by a cursor

Available in
PSQL

Syntax

FETCH [<fetch_scroll> FROM] cursor_name
[INTO [:]varname [, [:]varname ...]];

<fetch_scroll> ::= 
  NEXT | PRIOR | FIRST | LAST
  | RELATIVE n
  | ABSOLUTE n

Table 104. FETCH Statement Parameters
The **FETCH** statement fetches the first and successive rows from the result set of the cursor and assigns the column values to PSQL variables. The **FETCH** statement can be used only with a cursor declared with the **DECLARE CURSOR** statement.

Using the optional *fetch_scroll* part of the **FETCH** statement, you can specify in which direction and how many rows to advance the cursor position. The **NEXT** clause can be used for scrollable and forward-only cursors. Other clauses are only supported for scrollable cursors.

### The Scroll Options

**NEXT**
- moves the cursor one row forward; this is the default

**PRIOR**
- moves the cursor one record back

**FIRST**
- moves the cursor to the first record.

**LAST**
- moves the cursor to the last record

**RELATIVE** \( n \)
- moves the cursor \( n \) rows from the current position; positive numbers move forward, negative numbers move backwards; using zero (0) will not move the cursor, and **ROW_COUNT** will be set to zero as no new row was fetched.

#### Bug: Fetching First Row Using **RELATIVE**

In Firebird 3.0.7 and earlier, it is not possible to fetch the first row using **FETCH RELATIVE 1** immediately after opening the cursor. As a workaround, use **FETCH** (or **FETCH NEXT**) to fetch the first row.

This will be fixed in Firebird 3.0.8, see [CORE-6486](#).

**ABSOLUTE** \( n \)
- moves the cursor to the specified row; \( n \) is an integer expression, where 1 indicates the first row. For negative values, the absolute position is taken from the end of the result set, so -1 indicates the last row, -2 the second to last row, etc. A value of zero (0) will position before the first row.

#### Bug: Positioning Beyond the Bounds of the Cursor

In Firebird 3.0.7 and earlier, using **ABSOLUTE** and **RELATIVE**, it is possible to position
the cursor beyond the bounds of the result set—instead of immediately before the first row or immediately after the last row. Subsequent calls to FETCH RELATIVE will then require an offset large enough to move back within bounds, instead of just 1 or -1 to move to the first or last row.

This will be fixed in Firebird 3.0.8, see CORE-6487

The optional INTO clause gets data from the current row of the cursor and loads them into PSQL variables. If fetch moved beyond the bounds of the result set, the variables will be set to NULL.

It is also possible to use the cursor name as a variable of a row type (similar to OLD and NEW in triggers), allowing access to the columns of the result set (i.e. `cursor_name.columnname`).

**Rules for Cursor Variables**

- When accessing a cursor variable in a DML statement, the colon prefix can be added before the cursor name (i.e. `:cursor_name.columnname`) for disambiguation, similar to variables.

  The cursor variable can be referenced without colon prefix, but in that case, depending on the scope of the contexts in the statement, the name may resolve in the statement context instead of to the cursor (e.g. you select from a table with the same name as the cursor).

- Cursor variables are read-only

- In a FOR SELECT statement without an AS CURSOR clause, you must use the INTO clause. If an AS CURSOR clause is specified, the INTO clause is allowed, but optional; you can access the fields through the cursor instead.

- Reading from a cursor variable returns the current field values. This means that an UPDATE statement (with a WHERE CURRENT OF clause) will update not only the table, but also the fields in the cursor variable for subsequent reads. Executing a DELETE statement (with a WHERE CURRENT OF clause) will set all fields in the cursor variable to NULL for subsequent reads

- When the cursor is not positioned on a row—it is positioned before the first row, or after the last row—attempts to read from the cursor variable will result in error “Cursor `cursor_name` is not positioned in a valid record”

For checking whether all the rows of the result set have been fetched, the context variable ROW_COUNT returns the number of rows fetched by the statement. If a record was fetched, then ROW_COUNT is one (1), otherwise zero (0).

**FETCH Examples**

1. Using the FETCH statement:

   ```sql
   CREATE OR ALTER PROCEDURE GET_RELATIONS_NAMES
   RETURNS (RNAME CHAR(31))
   AS
   DECLARE C CURSOR FOR (SELECT RDB$RELATION_NAME FROM RDB$RELATIONS);
   BEGIN
   ```
OPEN C;
WHILE (1 = 1) DO
BEGIN
  FETCH C INTO RNAME;
  IF (ROW_COUNT = 0) THEN
    LEAVE;
    SUSPEND;
  END
CLOSE C;
END

2. Using the FETCH statement with nested cursors:

EXECUTE BLOCK
  RETURNS (SCRIPT BLOB SUB_TYPE TEXT)
AS
  DECLARE VARIABLE FIELDS VARCHAR (8191);
  DECLARE VARIABLE FIELD_NAME TYPE OF RDB$FIELD_NAME;
  DECLARE VARIABLE RELATION RDB$RELATION_NAME;
  DECLARE VARIABLE SRC TYPE OF COLUMN RDB$RELATIONS.RDB$VIEW_SOURCE;
  -- Named cursor declaration
  DECLARE VARIABLE CUR_R CURSOR FOR (
    SELECT
      RDB$RELATION_NAME,
      RDB$VIEW_SOURCE
    FROM RDB$RELATIONS
    WHERE RDB$VIEW_SOURCE IS NOT NULL);
  -- Declaring a named cursor in which
  -- a local variable is used
  DECLARE CUR_F CURSOR FOR (
    SELECT RDB$FIELD_NAME
    FROM RDB$RELATION_FIELDS
    WHERE
      -- It is important that the variable must be declared earlier
      RDB$RELATION_NAME =: RELATION);
BEGIN
  OPEN CUR_R;
  WHILE (1 = 1) DO
    BEGIN
      FETCH CUR_R INTO RELATION, SRC;
      IF (ROW_COUNT = 0) THEN
        LEAVE;
        FIELDS = NULL;
      -- Cursor CUR_F will use the value
      -- the RELATION variable initialized above
      OPEN CUR_F;
      WHILE (1 = 1) DO
        BEGIN
          FETCH CUR_F INTO FIELD_NAME;
          IF (ROW_COUNT = 0) THEN
          END
        END
      END
  END
END
LEAVE;
IF (FIELDS IS NULL) THEN
    FIELDS = TRIM(FIELD_NAME);
ELSE
    FIELDS = FIELDS || ', ' || TRIM(FIELD_NAME);
END
CLOSE CUR_F;
SCRIPT = 'CREATE VIEW' || RELATION;
IF (FIELDS IS NOT NULL) THEN
    SCRIPT = SCRIPT || '(' || FIELDS || ')
    SCRIPT = SCRIPT || 'AS' || ASCII_CHAR (13);
    SCRIPT = SCRIPT || SRC;
    SUSPEND;
END
CLOSE CUR_R;
EN

3. An example of using the FETCH statement with a scrollable cursor

EXECUTE BLOCK
    RETURNS (N INT, RNAME CHAR (31))
AS
    DECLARE C SCROLL CURSOR FOR (SELECT
        ROW_NUMBER() OVER (ORDER BY RDB$RELATION_NAME) AS N,
        RDB$RELATION_NAME
    FROM RDB$RELATIONS
    ORDER BY RDB$RELATION_NAME);
BEGIN
    OPEN C;
    -- move to the first record (N = 1)
    FETCH FIRST FROM C;
    RNAME = C.RDB$RELATION_NAME;
    N = C.N;
    SUSPEND;
    -- move 1 record forward (N = 2)
    FETCH NEXT FROM C;
    RNAME = C.RDB$RELATION_NAME;
    N = C.N;
    SUSPEND;
    -- move to the fifth record (N = 5)
    FETCH ABSOLUTE 5 FROM C;
    RNAME = C.RDB$RELATION_NAME;
    N = C.N;
    SUSPEND;
    -- move 1 record backward (N = 4)
    FETCH PRIOR FROM C;
    RNAME = C.RDB$RELATION_NAME;
    N = C.N;
    SUSPEND;
-- move 3 records forward (N = 7)
FETCH RELATIVE 3 FROM C;
   RNAME = C.RDB$RELATION_NAME;
   N = C.N;
   SUSPEND;
-- move back 5 records (N = 2)
FETCH RELATIVE -5 FROM C;
   RNAME = C.RDB$RELATION_NAME;
   N = C.N;
   SUSPEND;
-- move to the first record (N = 1)
FETCH FIRST FROM C;
   RNAME = C.RDB$RELATION_NAME;
   N = C.N;
   SUSPEND;
-- move to the last entry
FETCH LAST FROM C;
   RNAME = C.RDB$RELATION_NAME;
   N = C.N;
   SUSPEND;
CLOSE C;
END

See also
DECLARE .. CURSOR, OPEN, CLOSE

7.7.19. CLOSE

Used for
Closing a declared cursor

Available in
PSQL

Syntax

    CLOSE cursor_name;

Table 105. CLOSE Statement Parameter

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursor_name</td>
<td>Cursor name. A cursor with this name must be previously declared with a DECLARE .. CURSOR statement and opened by an OPEN statement</td>
</tr>
</tbody>
</table>

A CLOSE statement closes an open cursor. Any cursors that are still open will be automatically closed after the module code completes execution. Only a cursor that was declared with DECLARE .. CURSOR can be closed with a CLOSE statement.
CLOSE Examples

See FETCH Examples

See also

DECLARE .. CURSOR, OPEN, FETCH

7.7.20. IN AUTONOMOUS TRANSACTION

Used for
Executing a statement or a block of statements in an autonomous transaction

Available in
PSQL

Syntax

```
IN AUTONOMOUS TRANSACTION DO <compound_statement>
```

Table 106. IN AUTONOMOUS TRANSACTION Statement Parameter

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compound_statement</td>
<td>A single statement, or a block of statements</td>
</tr>
</tbody>
</table>

The IN AUTONOMOUS TRANSACTION statement enables execution of a statement or a block of statements in an autonomous transaction. Code running in an autonomous transaction will be committed right after its successful execution, regardless of the status of its parent transaction. This can be used when certain operations must not be rolled back, even if an error occurs in the parent transaction.

An autonomous transaction has the same isolation level as its parent transaction. Any exception that is thrown in the block of the autonomous transaction code will result in the autonomous transaction being rolled back and all changes made will be undone. If the code executes successfully, the autonomous transaction will be committed.

IN AUTONOMOUS TRANSACTION Examples

Using an autonomous transaction in a trigger for the database ON CONNECT event, in order to log all connection attempts, including those that failed:

```
CREATE TRIGGER TR_CONNECT ON CONNECT
AS
BEGIN
  -- Logging all attempts to connect to the database
  IN AUTONOMOUS TRANSACTION DO
    INSERT INTO LOG(MSG)
    VALUES ('USER ' || CURRENT_USER || ' CONNECTS.');
  IF (EXISTS(SELECT *
              FROM BLOCKED_USERS
              WHERE USERNAME = CURRENT_USER)) THEN
```

Chapter 7. Procedural SQL (PSQL) Statements

366
BEGIN
  -- Logging that the attempt to connect
  -- to the database failed and sending
  -- a message about the event
IN AUTONOMOUS TRANSACTION DO
BEGIN
  INSERT INTO LOG(MSG)
  VALUES ('USER ' || CURRENT_USER || ' REFUSED.');
  POST_EVENT 'CONNECTION ATTEMPT BY BLOCKED USER!';
END
  -- now calling an exception
EXCEPTION EX_BADUSER;
END
END

See also
Transaction Control

7.7.21. POST_EVENT

Used for
Notifying listening clients about database events in a module

Available in
PSQL

Syntax

POST_EVENT event_name;

Table 107. POST_EVENT Statement Parameter

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_name</td>
<td>Event name (message) limited to 127 bytes</td>
</tr>
</tbody>
</table>

The POST_EVENT statement notifies the event manager about the event, which saves it to an event table. When the transaction is committed, the event manager notifies applications that are signalling their interest in the event.

The event name can be some sort of code, or a short message: the choice is open as it is just a string up to 127 bytes.

The content of the string can be a string literal, a variable or any valid SQL expression that resolves to a string.

POST_EVENT Examples

Notifying the listening applications about inserting a record into the SALES table:
CREATE TRIGGER POST_NEW_ORDER FOR SALES
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
  POST_EVENT 'new_order';
END

7.7.22. RETURN

Used for
Return a value from a stored function

Available in
PSQL

Syntax

```
RETURN value;
```

Table 108. RETURN Statement Parameter

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Expression with the value to return; Can be any expression type-compatible with the return type of the function</td>
</tr>
</tbody>
</table>

The RETURN statement ends the execution of a function and returns the value of the expression value.

RETURN can only be used in PSQL functions (stored and local functions).

RETURN Examples

See CREATE FUNCTION Examples

7.8. Trapping and Handling Errors

Firebird has a useful lexicon of PSQL statements and resources for trapping errors in modules and for handling them. Firebird uses built-in exceptions that are raised for errors occurring when working DML and DDL statements.

In PSQL code, exceptions are handled by means of the WHEN statement. Handling an exception in the code involves either fixing the problem in situ, or stepping past it; either solution allows execution to continue without returning an exception message to the client.

An exception results in execution being terminated in the current block. Instead of passing the execution to the END statement, the procedure moves outward through levels of nested blocks, starting from the block where the exception is caught, searching for the code of the handler that “knows” about this exception. It stops searching when it finds the first WHEN statement that can
handle this exception.

### 7.8.1. System Exceptions

An exception is a message that is generated when an error occurs.

All exceptions handled by Firebird have predefined numeric values for context variables (symbols) and text messages associated with them. Error messages are output in English by default. Localized Firebird builds are available, where error messages are translated into other languages.

Complete listings of the system exceptions can be found in *Appendix B: Exception Codes and Messages*:

- SQLSTATE Error Codes and Descriptions
- "GDSCODE Error Codes, SQLCODEs and Descriptions"

### 7.8.2. Custom Exceptions

Custom exceptions can be declared in the database as persistent objects and called in the PSQL code to signal specific errors; for example, to enforce certain business rules. A custom exception consists of an identifier, and a default message of 1021 bytes. For details, see *CREATE EXCEPTION*.

### 7.8.3. EXCEPTION

*Used for*

Throwing a user-defined exception or re-throwing an exception

*Available in*

PSQL

*Syntax*

```
EXCEPTION [     
  exception_name   
  [ custom_message  
    | USING (<value_list>) ]  
]  

<value_list> ::= <val> [, <val> ...]
```

*Table 109. EXCEPTION Statement Parameters*

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception_name</td>
<td>Exception name</td>
</tr>
<tr>
<td>custom_message</td>
<td>Alternative message text to be returned to the caller interface when an exception is thrown. Maximum length of the text message is 1,021 bytes</td>
</tr>
<tr>
<td>val</td>
<td>Value expression that replaces parameter slots in the exception message text</td>
</tr>
</tbody>
</table>
The EXCEPTION statement with exception_name throws the user-defined exception with the specified name. An alternative message text of up to 1,021 bytes can optionally override the exception's default message text.

The default exception message can contain slots for parameters that can be filled when throwing an exception. To pass parameter values to an exception, use the USING clause. Considering, in left-to-right order, each parameter passed in the exception-raising statement as “the Nth”, with N starting at 1:

- If the Nth parameter is not passed, its slot is not replaced
- If a NULL parameter is passed, the slot will be replaced with the string “*** null ***”
- If more parameters are passed than are defined in the exception message, the surplus ones are ignored
- The maximum number of parameters is 9
- The maximum message length, including parameter values, is 1053 bytes

The status vector is generated this code combination isc_except, <exception number>, isc_formatted_exception, <formatted exception message>, <exception parameters>.

As a new error code (isc_formatted_exception) is used, the client must be version 3.0, or at least use the firebird.msg from version 3.0, in order to translate the status vector to a string.

If the message contains a parameter slot number that is greater than 9, the second and subsequent digits will be treated as literal text. For example @10 will be interpreted as slot 1 followed by a literal ‘0’.

As an example:

```sql
CREATE EXCEPTION ex1
    'something wrong in @ 1 @ 2 @ 3 @ 4 @ 5 @ 6 @ 7 @ 8 @ 9 @ 10 @ 11';
SET TERM ^;
EXECUTE BLOCK AS
BEGIN
    EXCEPTION ex1 USING ( 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' );
END^ 
```

This will produce the following output

```
Statement failed, SQLSTATE = HY000
exception 1
-EX1
-something wrong in abcdefghi a0 a1
```
Exceptions can be handled in a \texttt{WHEN} \ldots \texttt{DO} statement. If an exception is not handled in a module, then the effects of the actions executed inside this module are cancelled, and the caller program receives the exception (either the default text, or the custom text).

Within the exception-handling block — and only within it — the caught exception can be re-thrown by executing the \texttt{EXCEPTION} statement without parameters. If located outside the block, the re-thrown \texttt{EXCEPTION} call has no effect.

Custom exceptions are stored in the system table \texttt{RDB$EXCEPTIONS}.

\textbf{EXCEPTION Examples}

1. Throwing an exception upon a condition in the \texttt{SHIP\_ORDER} stored procedure:

```sql
CREATE OR ALTER PROCEDURE SHIP\_ORDER
  (PO\_NUM CHAR(8))
AS
  DECLARE VARIABLE ord\_stat CHAR(7);
  DECLARE VARIABLE hold\_stat CHAR(1);
  DECLARE VARIABLE cust\_no INTEGER;
  DECLARE VARIABLE any\_po CHAR(8);
BEGIN
  SELECT
    s.order\_status,
    c.on\_hold,
    c.cust\_no
  FROM
    sales s, customer c
  WHERE
    po\_number = :po\_num AND
    s.cust\_no = c.cust\_no
  INTO :ord\_stat,
       :hold\_stat,
       :cust\_no;

  IF (ord\_stat = 'shipped') THEN
    EXCEPTION order\_already\_shipped;
    /* Other statements */
  END
END
```

2. Throwing an exception upon a condition and replacing the original message with an alternative message:

```sql
CREATE OR ALTER PROCEDURE SHIP\_ORDER
  (PO\_NUM CHAR(8))
AS
  DECLARE VARIABLE ord\_stat CHAR(7);
  DECLARE VARIABLE hold\_stat CHAR(1);
  DECLARE VARIABLE cust\_no INTEGER;
```
DECLARE VARIABLE any_po    CHAR(8);
BEGIN
  SELECT
    s.order_status,
    c.on_hold,
    c.cust_no
  FROM
    sales s, customer c
  WHERE
    po_number = :po_num AND
    s.cust_no = c.cust_no
  INTO :ord_stat,
    :hold_stat,
    :cust_no;
  IF (ord_stat = 'shipped') THEN
    EXCEPTION order_already_shipped
    'Order status is " || ord_stat || "';
    /* Other statements */
  END
END

3. Using a parameterized exception:

CREATE EXCEPTION EX_BAD_SP_NAME
  'Name of procedures must start with '@ 1' ':' '@ 2' ' ;
...
CREATE TRIGGER TRG_SP_CREATE BEFORE CREATE PROCEDURE
AS
  DECLARE SP_NAME VARCHAR(255);
BEGIN
  SP_NAME = RDB$GET_CONTEXT ('DDL_TRIGGER', 'OBJECT_NAME');
  IF (SP_NAME NOT STARTING 'SP_') THEN
    EXCEPTION EX_BAD_SP_NAME USING ('SP_', SP_NAME);
  END

4. Logging an error and re-throwing it in the WHEN block:

CREATE PROCEDURE ADD_COUNTRY (ACountryName COUNTRYNAME,
  ACurrency VARCHAR(10))
AS
BEGIN
  INSERT INTO country (country, currency)
  VALUES (:ACountryName, :ACurrency);
  WHEN ANY DO
  BEGIN
    -- write an error in log

IN AUTONOMOUS TRANSACTION DO
INSERT INTO ERROR_LOG (PSQL_MODULE,
    GDS_CODE,
    SQL_CODE,
    SQL_STATE)
VALUES ('ADD_COUNTRY',
    GDS_CODE,
    SQL_CODE,
    SQLSTATE);
-- Re-throw exception
EXCEPTION;
END
END

See also
CREATE EXCEPTION, WHEN ... DO

7.8.4. WHEN ... DO

Used for
Catching an exception and handling the error

Available in
PSQL

Syntax

WHEN {<error> [, <error> ...] | ANY}
DO <compound_statement>

<error> ::= 
    { EXCEPTION exception_name
    | SQLCODE number
    | GDS_CODE errcode
    | SQLSTATE sqlstate_code }

Table 110. WHEN ... DO Statement Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception_name</td>
<td>Exception name</td>
</tr>
<tr>
<td>number</td>
<td>SQLCODE error code</td>
</tr>
<tr>
<td>errcode</td>
<td>Symbolic GDS_CODE error name</td>
</tr>
<tr>
<td>sqlstate_code</td>
<td>String literal with the SQLSTATE error code</td>
</tr>
<tr>
<td>compound_statement</td>
<td>A single statement, or a block of statements</td>
</tr>
</tbody>
</table>

The WHEN ... DO statement handles Firebird errors and user-defined exceptions. The statement catches all errors and user-defined exceptions listed after the keyword WHEN keyword. If WHEN is
followed by the keyword ANY, the statement catches any error or user-defined exception, even if they have already been handled in a WHEN block located higher up.

The WHEN … DO block must be located at the very end of a block of statements, before the block's END statement.

The keyword DO is followed by a statement, or a block of statements inside a BEGIN … END block, that handles the exception. The SQLCODE, GDSCODE, and SQLSTATE context variables are available in the context of this statement or block. The EXCEPTION statement, without parameters, can also be used in this context to re-throw the error or exception.

### Targeting GDSCODE

The argument for the WHEN GDSCODE clause is the symbolic name associated with the internally-defined exception, such as grant_obj_notfound for GDS error 335544551.

In statement or block of statements of the DO clause, a GDSCODE context variable, containing the numeric code, becomes available. That numeric code is required if you want to compare a GDSCODE exception with a targeted error. To compare it with a specific error, you need to use a numeric values, for example 335544551 for grant_obj_notfound.

Similar context variables are available for SQLCODE and SQLSTATE.

The WHEN … DO statement or block is only executed when one of the events targeted by its conditions occurs at run-time. If the WHEN … DO statement is executed, even if it actually does nothing, execution will continue as if no error occurred: the error or user-defined exception neither terminates nor rolls back the operations of the trigger or stored procedure.

However, if the WHEN … DO statement or block does nothing to handle or resolve the error, the DML statement (SELECT, INSERT, UPDATE, DELETE, MERGE) that caused the error will be rolled back and none of the statements below it in the same block of statements are executed.

1. If the error is not caused by one of the DML statements (SELECT, INSERT, UPDATE, DELETE, MERGE), the entire block of statements will be rolled back, not just the one that caused an error. Any operations in the WHEN … DO statement will be rolled back as well. The same limitation applies to the EXECUTE PROCEDURE statement. Read an interesting discussion of the phenomenon in Firebird Tracker ticket CORE-4483.

2. In selectable stored procedures, output rows that were already passed to the client in previous iterations of a FOR SELECT … DO … SUSPEND loop remain available to the client if an exception is thrown subsequently in the process of retrieving rows.

### Scope of a WHEN … DO Statement

A WHEN … DO statement catches errors and exceptions in the current block of statements. It also catches similar exceptions in nested blocks, if those exceptions have not been handled in those nested blocks.
All changes made before the statement that caused the error are visible to a WHEN … DO statement. However, if you try to log them in an autonomous transaction, those changes are unavailable, because the transaction where the changes took place is not committed at the point when the autonomous transaction is started. Example 4, below, demonstrates this behaviour.

When handling exceptions, it is sometimes desirable to handle the exception by writing a log message to mark the fault and having execution continue past the faulty record. Logs can be written to regular tables, but there is a problem with that: the log records will “disappear” if an unhandled error causes the module to stop executing, and a rollback is performed. Use of external tables can be useful here, as data written to them is transaction-independent. The linked external file will still be there, regardless of whether the overall process succeeds or not.

**Examples using WHEN…DO**

1. Replacing the standard error with a custom one:

   ```sql
   CREATE EXCEPTION COUNTRY_EXIST '';
   SET TERM ^;
   CREATE PROCEDURE ADD_COUNTRY ( 
       ACountryName COUNTRYNAME,
       ACurrency VARCHAR(10) )
   AS
   BEGIN
       INSERT INTO country (country, currency) 
       VALUES (:ACountryName, :ACurrency);

       WHEN SQLCODE -803 DO 
           EXCEPTION COUNTRY_EXIST 'Country already exists!';
   END^ 
   SET TERM ^;
   
   CREATE EXCEPTION COUNTRY_EXIST '';
   SET TERM ^;
   CREATE PROCEDURE ADD_COUNTRY ( 
       ACountryName COUNTRYNAME,
       ACurrency VARCHAR(10) )
   AS
   BEGIN
       INSERT INTO country (country, currency) 
       VALUES (:ACountryName, :ACurrency);

       WHEN SQLCODE -803 DO 
           EXCEPTION COUNTRY_EXIST 'Country already exists!';
   END^ 
   SET TERM ^;
   ```

2. Logging an error and re-throwing it in the WHEN block:

   ```sql
   CREATE PROCEDURE ADD_COUNTRY ( 
       ACountryName COUNTRYNAME,
       ACurrency VARCHAR(10) )
   AS
   BEGIN
       INSERT INTO country (country, currency) 
       VALUES (:ACountryName, :ACurrency);

       WHEN ANY DO 
       BEGIN
           -- write an error in log
           IN AUTONOMOUS TRANSACTION DO 
           INSERT INTO ERROR_LOG (PSQL_MODULE,
   ```
VALUES ('ADD_COUNTRY', GDSCODE, SQLCODE, SQLSTATE);

-- Re-throw exception
EXCEPTION;
END
END

3. Handling several errors in one WHEN block

... WHEN GDSCODE GRANT_OBJ_NOTFOUND,
    GDSCODE GRANT_FLD_NOTFOUND,
    GDSCODE GRANT_NOPRIV,
    GDSCODE GRANT_NOPRIV_ON_BASE
DO BEGIN
    EXECUTE PROCEDURE LOG_GRANT_ERROR(GDSCODE);
    EXIT;
END ...

4. Catching errors using the SQLSTATE code

EXECUTE BLOCK AS
    DECLARE VARIABLE I INT;
BEGIN BEGIN
    I = 1/0;
    WHEN SQLSTATE '22003' DO
        EXCEPTION E_CUSTOM_EXCEPTION
        'Numeric value out of range.';
    WHEN SQLSTATE '22012' DO
        EXCEPTION E_CUSTOM_EXCEPTION
        'Division by zero.';
    WHEN SQLSTATE '23000' DO
        EXCEPTION E_CUSTOM_EXCEPTION
        'Integrity constraint violation.';
END END

See also
EXCEPTION, CREATE EXCEPTION, SQLCODE and GDSCODE Error Codes and Message Texts and SQLSTATE
Codes and Message Texts, GDSCODE, SQLCODE, SQLSTATE
Chapter 8. Built-in Scalar Functions

Upgraders: PLEASE READ!

Many functions that were implemented as external functions (UDFs) in earlier versions of Firebird have been progressively re-implemented as internal (built-in) functions. If some external function of the same name as a built-in one is declared in your database, it will remain there and it will override any internal function of the same name.

To make the internal function available, you need either to DROP the UDF, or to use ALTER EXTERNAL FUNCTION to change the declared name of the UDF.

8.1. Context Functions

8.1.1. RDB$GET_CONTEXT()

Available in
DSQL, PSQL * As a declared UDF it should be available in ESQL

Result type
VARCHAR(255)

Syntax

RDB$GET_CONTEXT ('<namespace>', <varname>)

<namespace> ::= SYSTEM | USER_SESSION | USER_TRANSACTION | DDL_TRIGGER
<varname> ::= A case-sensitive quoted string of max. 80 characters

Table 111. RDB$GET_CONTEXT Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace</td>
<td>Namespace</td>
</tr>
<tr>
<td>varname</td>
<td>Variable name. Case-sensitive. Maximum length is 80 characters</td>
</tr>
</tbody>
</table>

Retrieves the value of a context variable from one of the namespaces SYSTEM, USER_SESSION and USER_TRANSACTION.

The namespaces

The USER_SESSION and USER_TRANSACTION namespaces are initially empty. The user can create and set variables in them with RDB$SET_CONTEXT() and retrieve them with RDB$GET_CONTEXT(). The SYSTEM namespace is read-only. The DDL_TRIGGER namespace is only valid in DDL triggers, and is read-only. It contains a number of predefined variables, shown below.

Return values and error behaviour

If the polled variable exists in the given namespace, its value will be returned as a string of max.
255 characters. If the namespace doesn't exist or if you try to access a non-existing variable in the SYSTEM namespace, an error is raised. If you request a non-existing variable in one of the other namespaces, NULL is returned. Both namespace and variable names must be given as single-quoted, case-sensitive, non-NULL strings.

**The SYSTEM Namespace**

*Context variables in the SYSTEM namespace*

**CLIENT_ADDRESS**
- For TCP, this is the IP address. For XNET, the local process ID. For all other protocols this variable is NULL.

**CLIENT_HOST**
- The wire protocol host name of remote client. Value is returned for all supported protocols.

**CLIENT_PID**
- Process ID of remote client application.

**CLIENT_PROCESS**
- Process name of remote client application.

**CURRENT_ROLE**
- Same as global `CURRENT_ROLE` variable.

**CURRENT_USER**
- Same as global `CURRENT_USER` variable.

**DB_NAME**
- Either the full path to the database or — if connecting via the path is disallowed — its alias.

**ENGINE_VERSION**
- The Firebird engine (server) version.

**ISOLATION_LEVEL**
- The isolation level of the current transaction: 'READ COMMITTED', 'SNAPSHOT' or 'CONSISTENCY'.

**LOCK_TIMEOUT**
- Lock timeout of the current transaction.

**NETWORK_PROTOCOL**
- The protocol used for the connection: 'TCPv4', 'TCPv6', 'WNET', 'XNET' or NULL.

**READ_ONLY**
- Returns 'TRUE' if current transaction is read-only and 'FALSE' otherwise.

**SESSION_ID**
- Same as global `CURRENT_CONNECTION` variable.
TRANSACTION_ID

Same as global CURRENT_TRANSACTION variable.

WIRE_COMPRESSED

Compression status of the current connection. If the connection is compressed, returns TRUE; if it is not compressed, returns FALSE. Returns NULL if the connection is embedded.

Introduced in Firebird 3.0.4.

WIRE_ENCRYPTED

Encryption status of the current connection. If the connection is encrypted, returns TRUE; if it is not encrypted, returns FALSE. Returns NULL if the connection is embedded.

Introduced in Firebird 3.0.4.

The DDL_TRIGGER Namespace

The DDL_TRIGGER namespace is valid only when a DDL trigger is running. Its use is also valid in stored procedures and functions called by DDL triggers.

The DDL_TRIGGER context works like a stack. Before a DDL trigger is fired, the values relative to the executed command are pushed onto this stack. After the trigger finishes, the values are popped. So in the case of cascade DDL statements, when a user DDL command fires a DDL trigger and this trigger executes another DDL command with EXECUTE STATEMENT, the values of the DDL_TRIGGER namespace are the ones relative to the command that fired the last DDL trigger on the call stack.

Context variables in the DDL_TRIGGER namespace

EVENT_TYPE

event type (CREATE, ALTER, DROP)

OBJECT_TYPE

object type (TABLE, VIEW, etc)

DDL_EVENT

event name (<ddl_event_item>), where <ddl_event_item> is EVENT_TYPE || ' ' || OBJECT_TYPE

OBJECT_NAME

metadata object name

OLD_OBJECT_NAME

for tracking the renaming of a domain (see note)

NEW_OBJECT_NAME

for tracking the renaming of a domain (see note)

SQL_TEXT

sql statement text

ALTER DOMAIN old-name TO new-name sets OLD_OBJECT_NAME and NEW_OBJECT_NAME in
both BEFORE and AFTER triggers. For this command, OBJECT_NAME will have the old object name in BEFORE triggers, and the new object name in AFTER triggers.

**Examples**

```sql
select rdb$get_context('SYSTEM', 'DB_NAME') from rdb$database
New.UserAddr = rdb$get_context('SYSTEM', 'CLIENT_ADDRESS');
insert into MyTable (TestField) values (rdb$get_context('USER_SESSION', 'MyVar'))
```

See also

RDB$SET_CONTEXT()

### 8.1.2. RDB$SET_CONTEXT()

**Available in**

DSQL, PSQL * As a declared UDF it should be available in ESQL

**Result type**

INTEGER

**Syntax**

```
RDB$SET_CONTEXT ('<namespace>', <varname>, <value> | NULL)
```

<table>
<thead>
<tr>
<th>&lt;namespace&gt;</th>
<th>USER_SESSION</th>
<th>USER_TRANSACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;varname&gt;</td>
<td>A case-sensitive quoted string of max. 80 characters</td>
<td></td>
</tr>
<tr>
<td>&lt;value&gt;</td>
<td>A value of any type, as long as it's castable to a VARCHAR(255)</td>
<td></td>
</tr>
</tbody>
</table>

**Table 112. RDB$SET_CONTEXT Function Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace</td>
<td>Namespace</td>
</tr>
<tr>
<td>varname</td>
<td>Variable name. Case-sensitive. Maximum length is 80 characters</td>
</tr>
<tr>
<td>value</td>
<td>Data of any type provided it can be cast to VARCHAR(255)</td>
</tr>
</tbody>
</table>

Creates, sets or unsets a variable in one of the user-writable namespaces USER_SESSION and USER_TRANSACTION.

**The namespaces**

The USER_SESSION and USER_TRANSACTION namespaces are initially empty. The user can create and set variables in them with RDB$SET_CONTEXT() and retrieve them with RDB$GET_CONTEXT(). The USER_SESSION context is bound to the current connection. Variables in USER_TRANSACTION only exist in the transaction in which they have been set. When the transaction ends, the context and all the
variables defined in it are destroyed.

Return values and error behaviour

The function returns 1 when the variable already existed before the call and 0 when it didn’t. To remove a variable from a context, set it to NULL. If the given namespace doesn’t exist, an error is raised. Both namespace and variable names must be entered as single-quoted, case-sensitive, non-NULL strings.

- The maximum number of variables in any single context is 1000.
- All USER_TRANSACTION variables will survive a ROLLBACK RETAIN (see ROLLBACK Options) or ROLLBACK TO SAVEPOINT unaltered, no matter at which point during the transaction they were set.
- Due to its UDF-like nature, RDB$SET_CONTEXT can — in PSQL only — be called like a void function, without assigning the result, as in the second example above. Regular internal functions don’t allow this type of use.

Examples

```
select rdb$set_context('USER_SESSION', 'MyVar', 493) from rdb$database

rdb$set_context('USER_SESSION', 'RecordsFound', RecCounter);

select rdb$set_context('USER_TRANSACTION', 'Savepoints', 'Yes')
    from rdb$database
```

See also

RDB$GET_CONTEXT()

8.2. Mathematical Functions

8.2.1. ABS()

Available in

DSQL, PSQL

Possible name conflict

YES → Read details

Result type

Numerical

Syntax

```
ABS (number)
```

Table 113. ABS Function Parameter
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

Returns the absolute value of the argument.

### 8.2.2. ACOS()

*Available in*  
DSQL, PSQL

*Possible name conflict*  
YES → Read details

*Result type*  
DOUBLE PRECISION

*Syntax*

\[
\text{ACOS (number)}
\]

*Table 114. ACOS Function Parameter*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type within the range ([-1, 1])</td>
</tr>
</tbody>
</table>

Returns the arc cosine of the argument.

- The result is an angle in the range \([0, \pi]\).

*See also*  
\[\text{COS}, \text{ASIN}, \text{ATAN}\]

### 8.2.3. ACOSH()

*Available in*  
DSQL, PSQL

*Result type*  
DOUBLE PRECISION

*Syntax*

\[
\text{ACOSH (number)}
\]

*Table 115. ACOSH Function Parameter*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Any non-NULL value in the range ([1, \infty]).</td>
</tr>
</tbody>
</table>
Returns the inverse hyperbolic cosine of the argument.

- The result is in the range [0, INF].

See also
COSH(), ASINH(), ATANH()

8.2.4. ASIN()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
DOUBLE PRECISION

Syntax

ASIN (number)

Table 116. ASIN Function Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type within the range [-1, 1]</td>
</tr>
</tbody>
</table>

Returns the arc sine of the argument.

- The result is an angle in the range [-pi/2, pi/2].

See also
SIN(), ACOS(), ATAN()

8.2.5. ASINH()

Available in
DSQL, PSQL

Result type
DOUBLE PRECISION

Syntax

ASINH (number)

Table 117. ASINH Function Parameter
Parameter | Description
---|---
number | Any non-NULL value in the range [-INF, INF].

Returns the inverse hyperbolic sine of the argument.

- The result is in the range [-INF, INF].

See also
SINH(), ACOSH(), ATANH()

### 8.2.6. ATAN()

**Available in**
DSQL, PSQL

**Possible name conflict**
YES → Read details

**Result type**
DOUBLE PRECISION

**Syntax**

```
ATAN (number)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

The function ATAN returns the arc tangent of the argument. The result is an angle in the range <\(-\pi/2, \pi/2\)>.

See also
ATAN2(), TAN(), ACOS(), ASIN()

### 8.2.7. ATAN2()

**Available in**
DSQL, PSQL

**Possible name conflict**
YES → Read details

**Result type**
DOUBLE PRECISION
Syntax

\[ \text{ATAN2}(y, x) \]

Table 119. ATAN2 Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>An expression of a numeric type</td>
</tr>
<tr>
<td>( x )</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

Returns the angle whose sine-to-cosine ratio is given by the two arguments, and whose sine and cosine signs correspond to the signs of the arguments. This allows results across the entire circle, including the angles \(-\pi/2\) and \(\pi/2\).

- The result is an angle in the range \([-\pi, \pi]\).
- If \(x\) is negative, the result is \(\pi\) if \(y\) is 0, and \(-\pi\) if \(y\) is -0.
- If both \(y\) and \(x\) are 0, the result is meaningless. Starting with Firebird 3.0, an error will be raised if both arguments are 0. At v.2.5.4, it is still not fixed in lower versions. For more details, visit Tracker ticket CORE-3201.

- A fully equivalent description of this function is the following: \(\text{ATAN2}(y, x)\) is the angle between the positive X-axis and the line from the origin to the point \((x, y)\). This also makes it obvious that \(\text{ATAN2}(0, 0)\) is undefined.
- If \(x\) is greater than 0, \(\text{ATAN2}(y, x)\) is the same as \(\text{ATAN}(y/x)\).
- If both sine and cosine of the angle are already known, \(\text{ATAN2}(\text{sin}, \text{cos})\) gives the angle.

8.2.8. ATANH()

Available in
DSQL, PSQL

Result type
DOUBLE PRECISION

Syntax

\[ \text{ATANH}(\text{number}) \]

Table 120. ATANH Function Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Any non-NULL value in the range &lt;-1, 1&gt;.</td>
</tr>
</tbody>
</table>

Returns the inverse hyperbolic tangent of the argument.
The result is a number in the range [-INF, INF].

See also
TANH(), ACOSH(), ASINH()

8.2.9. CEIL(), CEILING()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details (Affects CEILING only)

Result type
BIGINT for exact numeric number, or DOUBLE PRECISION for floating point number

Syntax

CEIL[ING] (number)

Table 121. CEIL[ING] Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

Returns the smallest whole number greater than or equal to the argument.

See also
FLOOR(), ROUND(), TRUNC()

8.2.10. COS()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
DOUBLE PRECISION

Syntax

COS (angle)

Table 122. COS Function Parameter
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>angle</td>
<td>An angle in radians</td>
</tr>
</tbody>
</table>

Returns an angle’s cosine. The argument must be given in radians.

- Any non-NULL result is — obviously — in the range [-1, 1].

See also

ACOS(), COT(), SIN(), TAN()

### 8.2.11. COSH()

**Available in**

DSQL, PSQL

**Possible name conflict**

YES → Read details

**Result type**

DOUBLE PRECISION

**Syntax**

COSH (number)

---

Table 123. COSH Function Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>A number of a numeric type</td>
</tr>
</tbody>
</table>

Returns the hyperbolic cosine of the argument.

- Any non-NULL result is in the range [1, INF].

See also

ACOSH(), SINH(), TANH()

### 8.2.12. COT()

**Available in**

DSQL, PSQL

**Possible name conflict**

YES → Read details

**Result type**

DOUBLE PRECISION
Chapter 8. Built-in Scalar Functions

**Syntax**

```plaintext
COT (angle)
```

**Table 124. COT Function Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>angle</td>
<td>An angle in radians</td>
</tr>
</tbody>
</table>

Returns an angle's cotangent. The argument must be given in radians.

*See also*

*COS(), SIN(), TAN()*

**8.2.13. EXP()**

*Available in*

DSQL, PSQL

*Result type*

DOUBLE PRECISION

**Syntax**

```plaintext
EXP (number)
```

**Table 125. EXP Function Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>A number of a numeric type</td>
</tr>
</tbody>
</table>

Returns the natural exponential, $e^{\text{number}}$

*See also*

*LN()*

**8.2.14. FLOOR()**

*Available in*

DSQL, PSQL

*Possible name conflict*

YES → Read details

*Result type*

BIGINT for exact numeric `number`, or DOUBLE PRECISION for floating point `number`
Chapter 8. Built-in Scalar Functions

Syntax

FLOOR (number)

Table 126. FLOOR Function Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

Returns the largest whole number smaller than or equal to the argument.

See also

CEIL(), CEILING(), ROUND(), TRUNC()

8.2.15. LN()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
DOUBLE PRECISION

Syntax

LN (number)

Table 127. LN Function Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

Returns the natural logarithm of the argument.

- An error is raised if the argument is negative or 0.

See also

EXP(), LOG(), LOG10()

8.2.16. LOG()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details
Result type
DOUBLE PRECISION

Syntax

\[ \text{LOG} (x, y) \]

Table 128. \text{LOG} Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Base. An expression of a numeric type</td>
</tr>
<tr>
<td>y</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

Returns the \( x \)-based logarithm of \( y \).

- If either argument is 0 or below, an error is raised. (Before 2.5, this would result in NaN, +/-INF or 0, depending on the exact values of the arguments.)
- If both arguments are 1, NaN is returned.
- If \( x = 1 \) and \( y < 1 \), -INF is returned.
- If \( x = 1 \) and \( y > 1 \), INF is returned.

See also
\text{POWER()}, \text{LN()}, \text{LOG10()}

8.2.17. \text{LOG10()}

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
DOUBLE PRECISION

Syntax

\[ \text{LOG10} (\text{number}) \]

Table 129. \text{LOG10} Function Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

Returns the 10-based logarithm of the argument.

- An error is raised if the argument is negative or 0. (In versions prior to 2.5, such values would
result in NaN and -INF, respectively.)

See also
POWER(), LN(), LOG()

8.2.18. MOD()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
SMALLINT, INTEGER or BIGINT depending on the type of a. If a is a floating-point type, the result is a BIGINT.

Syntax

MOD (a, b)

Table 130. MOD Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>An expression of a numeric type</td>
</tr>
<tr>
<td>b</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

Returns the remainder of an integer division.

- Non-integer arguments are rounded before the division takes place. So, “mod(7.5, 2.5)” gives 2 ("mod(8, 3)") not 0.

8.2.19. PI()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
DOUBLE PRECISION

Syntax

PI ()

Returns an approximation of the value of pi.
8.2.20. POWER()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
DOUBLE PRECISION

Syntax

```
POWER (x, y)
```

Table 131. POWER Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>An expression of a numeric type</td>
</tr>
<tr>
<td>y</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

Returns \( x \) to the power of \( y \) \( (x^y) \).

See also

EXP(), LOG(), LOG10(), SQRT()

8.2.21. RAND()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
DOUBLE PRECISION

Syntax

```
RAND ()
```

Returns a random number between 0 and 1.

8.2.22. ROUND()

Available in
DSQL, PSQL
**Possible name conflict**

YES → Read details

**Result type**

INTEGER, (scaled) BIGINT or DOUBLE PRECISION

**Syntax**

```
ROUND (number [, scale])
```

**Table 132. ROUND Function Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type</td>
</tr>
<tr>
<td>scale</td>
<td>An integer specifying the number of decimal places toward which rounding is to be performed, e.g.:</td>
</tr>
<tr>
<td></td>
<td>• 2 for rounding to the nearest multiple of 0.01</td>
</tr>
<tr>
<td></td>
<td>• 1 for rounding to the nearest multiple of 0.1</td>
</tr>
<tr>
<td></td>
<td>• 0 for rounding to the nearest whole number</td>
</tr>
<tr>
<td></td>
<td>• -1 for rounding to the nearest multiple of 10</td>
</tr>
<tr>
<td></td>
<td>• -2 for rounding to the nearest multiple of 100</td>
</tr>
</tbody>
</table>

Rounds a number to the nearest integer. If the fractional part is exactly 0.5, rounding is upward for positive numbers and downward for negative numbers. With the optional `scale` argument, the number can be rounded to powers-of-ten multiples (tens, hundreds, tenths, hundredths, etc.) instead of just integers.

⚠️ If you are used to the behaviour of the external function ROUND, please notice that the *internal* function always rounds halves away from zero, i.e. downward for negative numbers.

**ROUND Examples**

If the `scale` argument is present, the result usually has the same scale as the first argument:

- `ROUND(123.654, 1) -- returns 123.700 (not 123.7)`
- `ROUND(8341.7, -3) -- returns 8000.0 (not 8000)`
- `ROUND(45.1212, 0) -- returns 45.0000 (not 45)`

Otherwise, the result scale is 0:

- `ROUND(45.1212) -- returns 45`

**See also**
CEIL(), CEILING(), FLOOR(), TRUNC()

### 8.2.23. SIGN()

*Available in*
DSQL, PSQL

*Possible name conflict*
YES → Read details

*Result type*
SMALLINT

*Syntax*

```plaintext
SIGN (number)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

Returns the sign of the argument: -1, 0 or 1.

### 8.2.24. SIN()

*Available in*
DSQL, PSQL

*Possible name conflict*
YES → Read details

*Result type*
DOUBLE PRECISION

*Syntax*

```plaintext
SIN (angle)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>angle</td>
<td>An angle, in radians</td>
</tr>
</tbody>
</table>

Returns an angle’s sine. The argument must be given in radians.

- Any non-NULL result is — obviously — in the range [-1, 1].

*See also*
ASIN(), COS(), COT(), TAN()

8.2.25. SINH()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
DOUBLE PRECISION

Syntax

SINH (number)

Table 135. SINH Function Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

Returns the hyperbolic sine of the argument.

See also
ASINH(), COSH(), TANH()

8.2.26. SQRT()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
DOUBLE PRECISION

Syntax

SQRT (number)

Table 136. SQRT Function Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>

Returns the square root of the argument.
• If \( \text{number} \) is negative, an error is raised.

\textit{See also}\n
\texttt{POWER()}

\subsection{8.2.27. TAN()}

\textit{Available in}\n
DSQL, PSQL

\textit{Possible name conflict}\n
YES \DOT \textit{Read details}

\textbf{Result type}\n
DOUBLE PRECISION

\textbf{Syntax}\n
\begin{verbatim}
TAN (angle)
\end{verbatim}

\textit{Table 137. TAN Function Parameter}\n
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>angle</td>
<td>An angle, in radians</td>
</tr>
</tbody>
</table>

Returns an angle's tangent. The argument must be given in radians.

\textit{See also}\n
\texttt{ATAN(), ATAN2(), COS(), COT(), SIN(), TAN()}

\subsection{8.2.28. TANH()}

\textit{Available in}\n
DSQL, PSQL

\textit{Possible name conflict}\n
YES \DOT \textit{Read details}

\textbf{Result type}\n
DOUBLE PRECISION

\textbf{Syntax}\n
\begin{verbatim}
TANH (number)
\end{verbatim}

\textit{Table 138. TANH Function Parameters}\n
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type</td>
</tr>
</tbody>
</table>
Returns the hyperbolic tangent of the argument.

- Due to rounding, any non-NULL result is in the range [-1, 1] (mathematically, it’s <-1, 1>).

**See also**

`ATANH()`, `COSH()`, `TANH()`

### 8.2.29. TRUNC()

**Available in**

DSQL, PSQL

**Result type**

INTEGER, (scaled) BIGINT or DOUBLE PRECISION

**Syntax**

```
TRUNC (number [, scale])
```

**Table 139. TRUNC Function Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>An expression of a numeric type</td>
</tr>
<tr>
<td>scale</td>
<td>An integer specifying the number of decimal places toward which truncating is to be performed, e.g.:</td>
</tr>
<tr>
<td></td>
<td>- 2 for truncating to the nearest multiple of 0.01</td>
</tr>
<tr>
<td></td>
<td>- 1 for truncating to the nearest multiple of 0.1</td>
</tr>
<tr>
<td></td>
<td>- 0 for truncating to the nearest whole number</td>
</tr>
<tr>
<td></td>
<td>- -1 for truncating to the nearest multiple of 10</td>
</tr>
<tr>
<td></td>
<td>- -2 for truncating to the nearest multiple of 100</td>
</tr>
</tbody>
</table>

Returns the integer part of a number. With the optional `scale` argument, the number can be truncated to powers-of-ten multiples (tens, hundreds, tenths, hundredths, etc.) instead of just integers.

- If the `scale` argument is present, the result usually has the same scale as the first argument, e.g.
  - `TRUNC(789.2225, 2)` returns 789.2200 (not 789.22)
  - `TRUNC(345.4, -2)` returns 300.0 (not 300)
  - `TRUNC(-163.41, 0)` returns -163.00 (not -163)
- Otherwise, the result scale is 0:
  - `TRUNC(-163.41)` returns -163
If you are used to the behaviour of the external function TRUNCATE, please notice that the internal function TRUNC always truncates toward zero, i.e. upward for negative numbers.

See also
CEIL(), CEILING(), FLOOR(), ROUND()

8.3. String Functions

8.3.1. ASCII_CHAR()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
CHAR(1) CHARACTER SET NONE

Syntax

```
ASCII_CHAR (code)
```

Table 140. ASCII_CHAR Function Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>An integer within the range from 0 to 255</td>
</tr>
</tbody>
</table>

Returns the ASCII character corresponding to the number passed in the argument.

• If you are used to the behaviour of the ASCII_CHAR UDF, which returns an empty string if the argument is 0, please notice that the internal function correctly returns a character with ASCII code 0 here.

8.3.2. ASCII_VAL()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
SMALLINT
Syntax

```
ASCII_VAL (ch)
```

Table 141. ASCII_VAL Function Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch</td>
<td>A string of the [VAR]CHAR data type or a text BLOB with the maximum size of 32,767 bytes</td>
</tr>
</tbody>
</table>

Returns the ASCII code of the character passed in.

- If the argument is a string with more than one character, the ASCII code of the first character is returned.
- If the argument is an empty string, 0 is returned.
- If the argument is NULL, NULL is returned.
- If the first character of the argument string is multi-byte, an error is raised. (A bug in Firebird 2.1 - 2.1.3 and 2.5.0 causes an error to be raised if any character in the string is multi-byte. This is fixed in versions 2.1.4 and 2.5.1.)

8.3.3. BIT_LENGTH()

**Available in**

DSQL, PSQL

**Result type**

INTEGER

**Syntax**

```
BIT_LENGTH (string)
```

Table 142. BIT_LENGTH Function Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>An expression of a string type</td>
</tr>
</tbody>
</table>

Gives the length in bits of the input string. For multi-byte character sets, this may be less than the number of characters times 8 times the “formal” number of bytes per character as found in `RDB$CHARACTER_SETS`.

With arguments of type CHAR, this function takes the entire formal string length (i.e. the declared length of a field or variable) into account. If you want to obtain the “logical” bit length, not counting the trailing spaces, right-TRIM the argument before passing it to BIT_LENGTH.

**BLOB support**
Since Firebird 2.1, this function fully supports text BLOBs of any length and character set.

**BIT_LENGTH Examples**

```sql
select bit_length('Hello!') from rdb$database
-- returns 48

select bit_length(_iso8859_1 'Grüß di!') from rdb$database
-- returns 64: ü and ß take up one byte each in ISO8859_1

select bit_length
  (cast (_iso8859_1 'Grüß di!' as varchar(24) character set utf8))
from rdb$database
-- returns 80: ü and ß take up two bytes each in UTF8

select bit_length
  (cast (_iso8859_1 'Grüß di!' as char(24) character set utf8))
from rdb$database
-- returns 208: all 24 CHAR positions count, and two of them are 16-bit
```

See also

`OCTET_LENGTH()`, `CHAR_LENGTH()`, `CHARACTER_LENGTH()`

**8.3.4. CHAR_LENGTH(), CHARACTER_LENGTH()**

*Available in*

DSQL, PSQL

*Result type*

INTEGER

*Syntax*

```sql
CHAR_LENGTH (string)
| CHARACTER_LENGTH (string)
```

**Table 143. CHAR[ACTER]_LENGTH Function Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>An expression of a string type</td>
</tr>
</tbody>
</table>

Gives the length in characters of the input string.

- With arguments of type CHAR, this function returns the formal string length (i.e. the declared length of a field or variable). If you want to obtain the “logical” length, not counting the trailing spaces, right-TRIM the argument before passing it to CHAR[ACTER]_LENGTH.
- **BLOB support**: Since Firebird 2.1, this function fully supports text BLOBs of any length and character set.
CHAR_LENGTH Examples

```sql
select char_length('Hello!') from rdb$database
-- returns 6

select char_length(_iso8859_1 'Grüß di!') from rdb$database
-- returns 8

select char_length
  (cast (_iso8859_1 'Grüß di!' as varchar(24) character set utf8))
from rdb$database
-- returns 8; the fact that ü and ß take up two bytes each is irrelevant

select char_length
  (cast (_iso8859_1 'Grüß di!' as char(24) character set utf8))
from rdb$database
-- returns 24: all 24 CHAR positions count
```

See also
BIT_LENGTH(), OCTET_LENGTH()

8.3.5. HASH()

Available in
DSQL, PSQL

Result type
BIGINT

Syntax

```
HASH (string)
```

Table 144. HASH Function Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>An expression of a string type</td>
</tr>
</tbody>
</table>

Returns a hash value for the input string. This function fully supports text BLOBs of any length and character set.

8.3.6. LEFT()

Available in
DSQL, PSQL
Result type
VARCHAR or BLOB

Syntax

\[
\text{LEFT (string, length)}
\]

Table 145. LEFT Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>An expression of a string type</td>
</tr>
<tr>
<td>length</td>
<td>Integer expression. Defines the number of characters to return</td>
</tr>
</tbody>
</table>

Returns the leftmost part of the argument string. The number of characters is given in the second argument.

- This function fully supports text BLOBs of any length, including those with a multi-byte character set.
- If \( \text{string} \) is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(\( n \)) with \( n \) the length of the input string.
- If the \( \text{length} \) argument exceeds the string length, the input string is returned unchanged.
- If the \( \text{length} \) argument is not a whole number, bankers' rounding (round-to-even) is applied, i.e. 0.5 becomes 0, 1.5 becomes 2, 2.5 becomes 2, 3.5 becomes 4, etc.

See also

\text{RIGHT()}

8.3.7. \text{LOWER()}

Available in
DSQL, ESQL, PSQL

Possible name conflict
YES → Read details below

Result type
(VAR)CHAR or BLOB

Syntax

\[
\text{LOWER (string)}
\]

Table 146. LOWER Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>An expression of a string type</td>
</tr>
</tbody>
</table>
Returns the lower-case equivalent of the input string. The exact result depends on the character set. With ASCII or NONE for instance, only ASCII characters are lowercased; with OCTETS, the entire string is returned unchanged. Since Firebird 2.1 this function also fully supports text BLOBs of any length and character set.

**Name Clash**
Because LOWER is a reserved word, the internal function will take precedence even if the external function by that name has also been declared. To call the (inferior!) external function, use double-quotes and the exact capitalisation, as in "LOWER"(string).

**LOWER Examples**
```
select Sheriff from Towns
where lower(Name) = 'cooper''s valley'
```

See also
UPPER()

**8.3.8. LPAD()**

*Available in*
DSQL, PSQL

*Possible name conflict*
YES → Read details

*Result type*
VARCHAR or BLOB

*Syntax*
```
LPAD (str, endlen [, padstr])
```

*Table 147. LPAD Function Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>An expression of a string type</td>
</tr>
<tr>
<td>=endlen</td>
<td>Output string length</td>
</tr>
<tr>
<td>padstr</td>
<td>The character or string to be used to pad the source string up to the specified length. Default is space (“ ”)</td>
</tr>
</tbody>
</table>

Left-pads a string with spaces or with a user-supplied string until a given length is reached.

- This function fully supports text BLOBs of any length and character set.
- If `str` is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(`endlen`).
If `padstr` is given and equals '' (empty string), no padding takes place.

If `endlen` is less than the current string length, the string is truncated to `endlen`, even if `padstr` is the empty string.

In Firebird 2.1-2.1.3, all non-BLOB results were of type `VARCHAR(32765)`, which made it advisable to cast them to a more modest size. This is no longer the case.

When used on a BLOB, this function may need to load the entire object into memory. Although it does try to limit memory consumption, this may affect performance if huge BLOBs are involved.

**LPAD Examples**

```
lpad ('Hello', 12)               -- returns '       Hello'
lpad ('Hello', 12, '-')          -- returns '-------Hello'
lpad ('Hello', 12, '')           -- returns 'Hello'
lpad ('Hello', 12, 'abc')        -- returns 'abcabcaHello'
lpad ('Hello', 12, 'abcdefghij') -- returns 'abcdefgHello'
lpad ('Hello', 2)                -- returns 'He'
lpad ('Hello', 2, '-')           -- returns 'He'
lpad ('Hello', 2, '')            -- returns 'He'
```

See also

`RPAD()`

### 8.3.9. OCTET_LENGTH()

*Available in*

DSQL, PSQL

*Result type*

INTEGER

*Syntax*

```
OCTET_LENGTH (string)
```

*Table 148. OCTET_LENGTH Function Parameter*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>An expression of a string type</td>
</tr>
</tbody>
</table>

Gives the length in bytes (octets) of the input string. For multi-byte character sets, this may be less than the number of characters times the “formal” number of bytes per character as found in `RDB$CHARACTER_SETS`.

With arguments of type CHAR, this function takes the entire formal string length (i.e.
the declared length of a field or variable) into account. If you want to obtain the “logical” byte length, not counting the trailing spaces, right-TRIM the argument before passing it to OCTET_LENGTH.

**BLOB support**

Since Firebird 2.1, this function fully supports text BLOBs of any length and character set.

**OCTET_LENGTH Examples**

```sql
select octet_length('Hello!') from rdb$database
-- returns 6

select octet_length(_iso8859_1 'Grüß di!') from rdb$database
-- returns 8: ü and ß take up one byte each in ISO8859_1

select octet_length
  (cast (_iso8859_1 'Grüß di!' as varchar(24) character set utf8))
from rdb$database
-- returns 10: ü and ß take up two bytes each in UTF8

select octet_length
  (cast (_iso8859_1 'Grüß di!' as char(24) character set utf8))
from rdb$database
-- returns 26: all 24 CHAR positions count, and two of them are 2-byte
```

*See also*

BIT_LENGTH(), CHAR_LENGTH(), CHARACTER_LENGTH()

### 8.3.10. OVERLAY()

*Available in*

DSQL, PSQL

*Result type*

VARCHAR or BLOB

**Syntax**

```sql
OVERLAY (string PLACING replacement FROM pos [FOR length])
```

**Table 149. OVERLAY Function Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string into which the replacement takes place</td>
</tr>
<tr>
<td>replacement</td>
<td>Replacement string</td>
</tr>
<tr>
<td>pos</td>
<td>The position from which replacement takes place (starting position)</td>
</tr>
</tbody>
</table>
OVERLAY() overwrites part of a string with another string. By default, the number of characters removed from (overwritten in) the host string equals the length of the replacement string. With the optional fourth argument, a different number of characters can be specified for removal.

- This function supports BLOBs of any length.
- If `string` or `replacement` is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(n) with n the sum of the lengths of `string` and `replacement`.
- As usual in SQL string functions, `pos` is 1-based.
- If `pos` is beyond the end of `string`, `replacement` is placed directly after `string`.
- If the number of characters from `pos` to the end of `string` is smaller than the length of `replacement` (or than the `length` argument, if present), `string` is truncated at `pos` and `replacement` placed after it.
- The effect of a “FOR 0” clause is that `replacement` is simply inserted into `string`.
- If any argument is NULL, the result is NULL.
- If `pos` or `length` is not a whole number, bankers' rounding (round-to-even) is applied, i.e. 0.5 becomes 0, 1.5 becomes 2, 2.5 becomes 2, 3.5 becomes 4, etc.

⚠️ When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

**OVERLAY Examples**

<table>
<thead>
<tr>
<th>Description</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>The number of characters that are to be overwritten</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OVERLAY() Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>overlay ('Goodbye' placing 'Hello' from 2) -- returns 'GHelloe'</td>
</tr>
<tr>
<td>overlay ('Goodbye' placing 'Hello' from 5) -- returns 'GoodHello'</td>
</tr>
<tr>
<td>overlay ('Goodbye' placing 'Hello' from 8) -- returns 'GoodbyeHello'</td>
</tr>
<tr>
<td>overlay ('Goodbye' placing 'Hello' from 20) -- returns 'GoodbyeHello'</td>
</tr>
<tr>
<td>overlay ('Goodbye' placing 'Hello' from 2 for 0) -- r. 'GHellooodbye'</td>
</tr>
<tr>
<td>overlay ('Goodbye' placing 'Hello' from 2 for 3) -- r. 'GHellobye'</td>
</tr>
<tr>
<td>overlay ('Goodbye' placing 'Hello' from 2 for 6) -- r. 'GHello'</td>
</tr>
<tr>
<td>overlay ('Goodbye' placing 'Hello' from 2 for 9) -- r. 'GHello'</td>
</tr>
<tr>
<td>overlay ('Goodbye' placing '' from 4) -- returns 'Goodbye'</td>
</tr>
<tr>
<td>overlay ('Goodbye' placing '' from 4 for 3) -- returns 'Gooe'</td>
</tr>
<tr>
<td>overlay ('Goodbye' placing '' from 4 for 20) -- returns 'Goo'</td>
</tr>
<tr>
<td>overlay ('' placing 'Hello' from 4) -- returns 'Hello'</td>
</tr>
<tr>
<td>overlay ('' placing 'Hello' from 4 for 0) -- returns 'Hello'</td>
</tr>
<tr>
<td>overlay ('' placing 'Hello' from 4 for 20) -- returns 'Hello'</td>
</tr>
</tbody>
</table>

See also

REPLACE()
8.3.11. POSITION()

Available in
DSQL, PSQL

Result type
INTEGER

Syntax

| POSITION (substr IN string) |
| POSITION (substr, string [, startpos]) |

Table 150. POSITION Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>substr</td>
<td>The substring whose position is to be searched for</td>
</tr>
<tr>
<td>string</td>
<td>The string which is to be searched</td>
</tr>
<tr>
<td>startpos</td>
<td>The position in string where the search is to start</td>
</tr>
</tbody>
</table>

Returns the (1-based) position of the first occurrence of a substring in a host string. With the optional third argument, the search starts at a given offset, disregarding any matches that may occur earlier in the string. If no match is found, the result is 0.

- The optional third argument is only supported in the second syntax (comma syntax).
- The empty string is considered a substring of every string. Therefore, if substr is ' ' (empty string) and string is not NULL, the result is:
  - 1 if startpos is not given;
  - startpos if startpos lies within string;
  - 0 if startpos lies beyond the end of string.

Notice: A bug in Firebird 2.1 - 2.1.3 and 2.5.0 causes POSITION to always return 1 if substr is the empty string. This is fixed in 2.1.4 and 2.5.1.

- This function fully supports text BLOBs of any size and character set.

When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

POSITION Examples

position ('be' in 'To be or not to be')   -- returns 4
position ('be', 'To be or not to be')     -- returns 4
position ('be', 'To be or not to be', 4)  -- returns 4
position ('be', 'To be or not to be', 8) -- returns 17
position ('be', 'To be or not to be', 18) -- returns 0
position ('be' in 'Alas, poor Yorick!') -- returns 0

See also

SUBSTRING()

8.3.12. REPLACE()

Available in
DSQL, PSQL

Result type
VARCHAR or BLOB

Syntax

REPLACE (str, find, repl)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>The string in which the replacement is to take place</td>
</tr>
<tr>
<td>find</td>
<td>The string to search for</td>
</tr>
<tr>
<td>repl</td>
<td>The replacement string</td>
</tr>
</tbody>
</table>

Replaces all occurrences of a substring in a string.

- This function fully supports text BLOBs of any length and character set.
- If any argument is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(n) with n calculated from the lengths of str, find and repl in such a way that even the maximum possible number of replacements won't overflow the field.
- If find is the empty string, str is returned unchanged.
- If repl is the empty string, all occurrences of find are deleted from str.
- If any argument is NULL, the result is always NULL, even if nothing would have been replaced.

⚠️ When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

REPLACE Examples

replace ('Billy Wilder', 'il', 'oog') -- returns 'Boogly Woogder'
replace ('Billy Wilder', 'il', '') -- returns 'Bly Wder'
replace ('Billy Wilder', null, 'oog') -- returns NULL
replace ('Billy Wilder', 'il', null) -- returns NULL
8.3.13. REVERSE()

*Available in*
DSQL, PSQL

*Result type*
VARCHAR

*Syntax*
```
REVERSE (string)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>An expression of a string type</td>
</tr>
</tbody>
</table>

Returns a string backwards.

**REVERSE Examples**

```
reverse ('spoonful')            -- returns 'lufnoops'
reverse ('Was it a cat I saw?') -- returns '?was I tac a ti saW'
```

This function comes in very handy if you want to group, search or order on string endings, e.g. when dealing with domain names or email addresses:

```
create index ix_people_email on people
computed by (reverse(email));

select * from people
where reverse(email) starting with reverse('.br');
```

8.3.14. RIGHT()

*Available in*
DSQL, PSQL

*Possible name conflict*
Result type
VARCHAR or BLOB

Syntax

```
RIGHT (string, length)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>An expression of a string type</td>
</tr>
<tr>
<td>length</td>
<td>Integer. Defines the number of characters to return</td>
</tr>
</tbody>
</table>

Returns the rightmost part of the argument string. The number of characters is given in the second argument.

- This function supports text BLOBs of any length, but has a bug in versions 2.1 - 2.1.3 and 2.5.0 that makes it fail with text BLOBs larger than 1024 bytes that have a multi-byte character set. This has been fixed in versions 2.1.4 and 2.5.1.
- If `string` is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR\((n)\) with \(n\) the length of the input string.
- If the `length` argument exceeds the string length, the input string is returned unchanged.
- If the `length` argument is not a whole number, bankers' rounding (round-to-even) is applied, i.e. 0.5 becomes 0, 1.5 becomes 2, 2.5 becomes 2, 3.5 becomes 4, etc.

⚠️ When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

See also

`LEFT()`, `SUBSTRING()`

8.3.15. **RPAD()**

Available in

DSQL, PSQL

Possible name conflict

YES → Read details

Result type

VARCHAR or BLOB
Syntax

RPAD (str, endlen [, padstr])

Table 154. RPAD Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>An expression of a string type</td>
</tr>
<tr>
<td>endlen</td>
<td>Output string length</td>
</tr>
<tr>
<td>endlen</td>
<td>The character or string to be used to pad the source string up to the specified length. Default is space (' ')</td>
</tr>
</tbody>
</table>

Right-pads a string with spaces or with a user-supplied string until a given length is reached.

- This function fully supports text BLOBs of any length and character set.
- If `str` is a BLOB, the result is a BLOB. Otherwise, the result is a VARCHAR(`endlen`).
- If `padstr` is given and equals '' (empty string), no padding takes place.
- If `endlen` is less than the current string length, the string is truncated to `endlen`, even if `padstr` is the empty string.

In Firebird 2.1-2.1.3, all non-BLOB results were of type VARCHAR(32765), which made it advisable to cast them to a more modest size. This is no longer the case.

When used on a BLOB, this function may need to load the entire object into memory. Although it does try to limit memory consumption, this may affect performance if huge BLOBs are involved.

RPAD Examples

rpad ('Hello', 12)          -- returns 'Hello       '
rpad ('Hello', 12, '-')     -- returns 'Hello-------'
rpad ('Hello', 12, '')      -- returns 'Hello'
rpad ('Hello', 12, 'abc')   -- returns 'Helloabcabca'
rpad ('Hello', 12, 'abcdefghij') -- returns 'Helloabcdefg'
rpad ('Hello', 2)           -- returns 'He'
rpad ('Hello', 2, '-')      -- returns 'He'
rpad ('Hello', 2, '')       -- returns 'He'

See also

LPAD()

8.3.16. SUBSTRING()

Available in
DSQL, PSQL
Result types
VARCHAR or BLOB

Syntax

```
SUBSTRING ( <substring-args> )

<substring-args> ::= str FROM startpos [FOR length] | str SIMILAR <similar-pattern> ESCAPE <escape>

Table 155. SUBSTRING Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>An expression of a string type</td>
</tr>
<tr>
<td>startpos</td>
<td>Integer expression, the position from which to start retrieving the substring</td>
</tr>
<tr>
<td>length</td>
<td>The number of characters to retrieve after the startpos</td>
</tr>
<tr>
<td>similar-pattern</td>
<td>SQL regular expression pattern to search for the substring</td>
</tr>
<tr>
<td>escape</td>
<td>Escape character</td>
</tr>
</tbody>
</table>
```

Returns a string’s substring starting at the given position, either to the end of the string or with a given length, or extracts a substring using an SQL regular expression pattern.

If any argument is NULL, the result is also NULL.

⚠️ When used on a BLOB, this function may need to load the entire object into memory. Although it does try to limit memory consumption, this may affect performance if huge BLOBs are involved.

**Positional SUBSTRING**

In its simple, positional form (with FROM), this function returns the substring starting at character position `startpos` (the first position being 1). Without the FOR argument, it returns all the remaining characters in the string. With FOR, it returns `length` characters or the remainder of the string, whichever is shorter.

The function fully supports binary and text BLOBs of any length, and with any character set. If `str` is a BLOB, the result is also a BLOB. For any other argument type, the result is a VARCHAR.

For non-BLOB arguments, the width of the result field is always equal to the length of `str`, regardless of `startpos` and `length`. So, `substring('pinhead' from 4 for 2)` will return a VARCHAR(7) containing
the string 'he'.

Example

```sql
insert into AbbrNames(AbbrName)
    select substring(LongName from 1 for 3) from LongNames
```

**Regular Expression SUBSTRING**

In the regular expression form (with SIMILAR), the SUBSTRING function returns part of the string matching an SQL regular expression pattern. If no match is found, NULL is returned.

The SIMILAR pattern is formed from three SQL regular expression patterns, \( R_1 \), \( R_2 \) and \( R_3 \). The entire pattern takes the form of \( R_1 | | '<\text{escape}>'' | | R_2 | | '<\text{escape}>'' | | R_3 \), where \(<\text{escape}>\) is the escape character defined in the ESCAPE clause. \( R_2 \) is the pattern that matches the substring to extract, and is enclosed between escaped double quotes (\(<\text{escape}>\)’, e.g. “#”’ with escape character ‘#’). \( R_1 \) matches the prefix of the string, and \( R_3 \) the suffix of the string. Both \( R_1 \) and \( R_3 \) are optional (they can be empty), but the pattern must match the entire string. In other words, it is not sufficient to specify a pattern that only finds the substring to extract.

The escaped double quotes around \( R_2 \) can be compared to defining a single capture group in more common regular expression syntax like PCRE. That is, the full pattern is equivalent to \( R_1(R_2)R_3 \), which must match the entire input string, and the capture group is the substring to be returned.

If any one of \( R_1 \), \( R_2 \), or \( R_3 \) is not a zero-length string and does not have the format of an SQL regular expression, then an exception is raised.

The full SQL regular expression format is described in [Syntax: SQL Regular Expressions](#).

**Examples**

```sql
substring('abcabc' similar 'a#"bcabc"c' escape '#')  -- bcab
substring('abcabc' similar 'a#"%#c' escape '#')     -- bcab
substring('abcabc' similar '_#"%#_' escape '#')     -- bcab
substring('abcabc' similar '#"(abc)*#"' escape '#')  -- abcabc
substring('abcabc' similar '#"abc#"' escape '#')     -- <null>
```

**See also**

POSITION(), LEFT(), RIGHT(), CHAR_LENGTH(), CHARACTER_LENGTH(), SIMILAR TO

**8.17. TRIM()**

*Available in*

DSQL, PSQL

*Result type*

VARCHAR or BLOB
Syntax

```
TRIM ([<adjust>] str)
```

```
<adjust> ::= {[[<where>] [what]] FROM
```

```
<where> ::= BOTH | LEADING | TRAILING
```

Table 156. TRIM Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>An expression of a string type</td>
</tr>
<tr>
<td>where</td>
<td>The position the substring is to be removed from — BOTH</td>
</tr>
<tr>
<td>what</td>
<td>The substring that should be removed (multiple times if there are several matches) from the beginning, the end, or both sides of the input string str. By default it is space (’ ’)</td>
</tr>
</tbody>
</table>

Removes leading and/or trailing spaces (or optionally other strings) from the input string. Since Firebird 2.1 this function fully supports text BLOBs of any length and character set.

- If str is a BLOB, the result is a BLOB. Otherwise, it is a VARCHAR(n) with n the formal length of str.
- Since Firebird 3.0, the maximum size of what — if a `BLOB` — was increased to 4GB, in previous versions the value of what could not exceed 32767 bytes.

When used on a BLOB, this function may need to load the entire object into memory. This may affect performance if huge BLOBs are involved.

**TRIM Examples**

```
select trim (' Waste no space ' ) from rdb$database
-- returns 'Waste no space'
```

```
select trim (leading from ' Waste no space ' ) from rdb$database
-- returns 'Waste no space '
```

```
select trim (leading '.' from ' Waste no space ' ) from rdb$database
-- returns ' Waste no space '
```

```
select trim (trailing '!' from 'Help!!!!') from rdb$database
-- returns 'Help'
```

```
select trim ('la' from 'lalala I love you Ella') from rdb$database
-- returns ' I love you El'
```

```
select trim ('la' from 'Lalala I love you Ella') from rdb$database
```
8.3.18. UPPER()

*Available in*
DSQL, ESQL, PSQL

*Result type*
(VAR)CHAR or BLOB

*Syntax*

```
UPPER (str)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>An expression of a string type</td>
</tr>
</tbody>
</table>

Returns the upper-case equivalent of the input string. The exact result depends on the character set. With ASCII or NONE for instance, only ASCII characters are uppercased; with OCTETS, the entire string is returned unchanged. Since Firebird 2.1 this function also fully supports text BLOBs of any length and character set.

**UPPER Examples**

```sql
select upper(_iso8859_1 'Débâcle')
from rdb$database
-- returns 'DÉBÂCLE' (before Firebird 2.0: 'DéBâCLE')

select upper(_iso8859_1 'Débâcle' collate fr_fr)
from rdb$database
-- returns 'DEBACLE', following French uppercasing rules
```

*See also*

LOWER()

8.4. Date and Time Functions

8.4.1. DATEADD()
Syntax

```
DATEADD (<args>)
```

```
<args> ::= 
    <amount> <unit> TO <datetime>
  | <unit>, <amount>, <datetime>
```

```
<amount> ::= an integer expression (negative to subtract)
```

```
<unit> ::= 
    YEAR | MONTH | WEEK | DAY
  | HOUR | MINUTE | SECOND | MILLISECOND
```

```
<datetime> ::= a DATE, TIME or TIMESTAMP expression
```

Table 158. DATEADD Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>An integer expression of the SMALLINT, INTEGER or BIGINT type. For unit MILLISECOND, the type is NUMERIC(18, 1). A negative value is subtracted.</td>
</tr>
<tr>
<td>unit</td>
<td>Date/time unit</td>
</tr>
<tr>
<td>datetime</td>
<td>An expression of the DATE, TIME or TIMESTAMP type</td>
</tr>
</tbody>
</table>

Adds the specified number of years, months, weeks, days, hours, minutes, seconds or milliseconds to a date/time value.

- The result type is determined by the third argument.
- With TIMESTAMP and DATE arguments, all units can be used.
- With TIME arguments, only HOUR, MINUTE, SECOND and MILLISECOND can be used.

Examples of DATEADD

```
dateadd (28 day to current_date)
dateadd (-6 hour to current_time)
dateadd (month, 9, DateOfConception)
dateadd (-38 week to DateOfBirth)
dateadd (minute, 90, time 'now')
dateadd (? year to date '11-Sep-1973')
```

```
select 
    cast(dateadd(-1 * extract(millisecond from ts) millisecond to ts) as varchar(30)) as t, 
    extract(millisecond from ts) as ms 
from ( 
    select timestamp '2014-06-09 13:50:17.4971' as ts 
    from rdb$database 
) a
```
See also
DATEDIFF(), Operations Using Date and Time Values

8.4.2. DATEDIFF()

Available in
DSQL, PSQL

Result type
BIGINT, or — since Firebird 3.0.8 — NUMERIC(18,1) for MILLISECOND

Syntax

DATEDIFF (<args>)

<args> ::=  
   <unit> FROM <moment1> TO <moment2>  
   | <unit>, <moment1>, <moment2>

<unit> ::=  
   YEAR | MONTH | WEEK | DAY  
   | HOUR | MINUTE | SECOND | MILLISECOND

<momentN> ::= a DATE, TIME or TIMESTAMP expression

Table 159. DATEDIFF Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>Date/time unit</td>
</tr>
<tr>
<td>moment1</td>
<td>An expression of the DATE, TIME or TIMESTAMP type</td>
</tr>
<tr>
<td>moment2</td>
<td>An expression of the DATE, TIME or TIMESTAMP type</td>
</tr>
</tbody>
</table>

Returns the number of years, months, weeks, days, hours, minutes, seconds or milliseconds elapsed between two date/time values. (The WEEK unit is new in 2.5.)

- DATE and TIMESTAMP arguments can be combined. No other mixes are allowed.
- With TIMESTAMP and DATE arguments, all units can be used. (Prior to Firebird 2.5, units smaller than DAY were disallowed for DATEs.)
- With TIME arguments, only HOUR, MINUTE, SECOND and MILLISECOND can be used.

Computation

- DATEDIFF doesn’t look at any smaller units than the one specified in the first argument. As a result,
- `datediff (year, date '1-Jan-2009', date '31-Dec-2009')` returns 0, but
- `datediff (year, date '31-Dec-2009', date '1-Jan-2010')` returns 1
  - It does, however, look at all the bigger units. So:
    - `datediff (day, date '26-Jun-1908', date '11-Sep-1973')` returns 23818
  - A negative result value indicates that `moment2` lies before `moment1`.

**DATEDIFF Examples**

- `datediff (hour from current_timestamp to timestamp '12-Jun-2059 06:00')`
- `datediff (minute from time '0:00' to current_time)`
- `datediff (month, current_date, date '1-1-1900')`
- `datediff (day from current_date to cast(? as date))`

*See also*

**DATEADD(), Operations Using Date and Time Values**

### 8.4.3. EXTRACT()

*Available in*

DSQL, ESQL, PSQL

*Result type*

SMALLINT or NUMERIC

*Syntax*

```sql
EXTRACT (<part> FROM <datetime>)
```

```sql
<part> ::= 
    YEAR | MONTH | WEEK 
    | DAY | WEEKDAY | YEARDAY 
    | HOUR | MINUTE | SECOND | MILLISECOND  
<datetime> ::= a DATE, TIME or TIMESTAMP expression
```

*Table 160. EXTRACT Function Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>part</td>
<td>Date/time unit</td>
</tr>
<tr>
<td>datetime</td>
<td>An expression of the DATE, TIME or TIMESTAMP type</td>
</tr>
</tbody>
</table>

Extracts and returns an element from a DATE, TIME or TIMESTAMP expression. This function was already added in InterBase 6, but not documented in the *Language Reference* at the time.

*Returned Data Types and Ranges*

The returned data types and possible ranges are shown in the table below. If you try to extract a
part that isn’t present in the date/time argument (e.g. SECOND from a DATE or YEAR from a TIME), an error occurs.

Table 161. Types and ranges of EXTRACT results

<table>
<thead>
<tr>
<th>Part</th>
<th>Type</th>
<th>Range</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>SMALLINT</td>
<td>1-9999</td>
<td></td>
</tr>
<tr>
<td>MONTH</td>
<td>SMALLINT</td>
<td>1-12</td>
<td></td>
</tr>
<tr>
<td>WEEK</td>
<td>SMALLINT</td>
<td>1-53</td>
<td></td>
</tr>
<tr>
<td>DAY</td>
<td>SMALLINT</td>
<td>1-31</td>
<td></td>
</tr>
<tr>
<td>WEEKDAY</td>
<td>SMALLINT</td>
<td>0-6</td>
<td>0 = Sunday</td>
</tr>
<tr>
<td>YEARDAY</td>
<td>SMALLINT</td>
<td>0-365</td>
<td>0 = January 1</td>
</tr>
<tr>
<td>HOUR</td>
<td>SMALLINT</td>
<td>0-23</td>
<td></td>
</tr>
<tr>
<td>MINUTE</td>
<td>SMALLINT</td>
<td>0-59</td>
<td></td>
</tr>
<tr>
<td>SECOND</td>
<td>NUMERIC(9,4)</td>
<td>0.0000-59.9999</td>
<td>includes millisecond as fraction</td>
</tr>
<tr>
<td>MILLISECOND</td>
<td>NUMERIC(9,1)</td>
<td>0.0-999.9</td>
<td>broken in 2.1, 2.1.1</td>
</tr>
</tbody>
</table>

**MILLISECOND**

Firebird 2.1 and up support extraction of the millisecond from a TIME or TIMESTAMP. The datatype returned is NUMERIC(9,1).

If you extract the millisecond from CURRENT_TIME, be aware that this variable defaults to seconds precision, so the result will always be 0. Extract from CURRENT_TIME(3) or CURRENT_TIMESTAMP to get milliseconds precision.

**WEEK**

Firebird 2.1 and up support extraction of the ISO-8601 week number from a DATE or TIMESTAMP. ISO-8601 weeks start on a Monday and always have the full seven days. Week 1 is the first week that has a majority (at least 4) of its days in the new year. The first 1-3 days of the year may belong to the last week (52 or 53) of the previous year. Likewise, a year’s final 1-3 days may belong to week 1 of the following year.

Be careful when combining WEEK and YEAR results. For instance, 30 December 2008 lies in week 1 of 2009, so extract(week from date '30 Dec 2008') returns 1. However, extracting YEAR always gives the calendar year, which is 2008. In this case, WEEK and YEAR are at odds with each other. The same happens when the first days of January belong to the last week of the previous year.

Please also notice that WEEKDAY is *not* ISO-8601 compliant: it returns 0 for Sunday, whereas ISO-8601 specifies 7.

See also

Data Types for Dates and Times
8.5. Type Casting Functions

8.5.1. CAST()

Available in
DSQL, ESQL, PSQL

Result type
As specified by target_type

Syntax

\[
\text{CAST (} \langle \text{expression} \rangle \ \text{AS} \ \langle \text{target\_type} \rangle \text{)}
\]

\[
\langle \text{target\_type} \rangle :\!::= \langle \text{domain\_or\_non\_array\_type} \rangle \ | \ \langle \text{array\_datatype} \rangle
\]

\[
\langle \text{domain\_or\_non\_array\_type} \rangle :\!::=
\]

!! See Scalar Data Types Syntax !!

\[
\langle \text{array\_datatype} \rangle :\!::=
\]

!! See Array Data Types Syntax !!

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>SQL expression</td>
</tr>
<tr>
<td>sql_datatype</td>
<td>SQL data type</td>
</tr>
</tbody>
</table>

CAST converts an expression to the desired datatype or domain. If the conversion is not possible, an error is raised.

Casting BLOBs
Successful casting to and from BLOBs is possible since Firebird 2.1.

“Shorthand” Syntax

Alternative syntax, supported only when casting a string literal to a DATE, TIME or TIMESTAMP:

\[
\text{datatype 'date/timestring'}
\]

This syntax was already available in InterBase, but was never properly documented. In the SQL standard, this feature is called “datetime literals”.

The short syntax is evaluated immediately at parse time, causing the value to stay the same until the statement is unprepared. For datetime literals like '12-Oct-2012' this makes no difference. For the pseudo-variables 'NOW', 'YESTERDAY', 'TODAY' and 'TOMORROW', this may not be what you want. If you need the value to be evaluated at
every call, use the full `CAST()` syntax.

Firebird 4 will disallow the use of 'NOW', 'YESTERDAY' and 'TOMORROW' in the shorthand cast, and only allow literals defining a fixed moment in time.

### Allowed Type Conversions

The following table shows the type conversions possible with `CAST`.

**Table 163. Possible Type-castings with `CAST`**

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric types</td>
<td>Numeric types</td>
</tr>
<tr>
<td>[VAR]CHAR</td>
<td>[VAR]CHAR</td>
</tr>
<tr>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>Datum types</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>[VAR]CHAR</td>
</tr>
<tr>
<td>TIME</td>
<td>BLOB</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
</tr>
</tbody>
</table>

Keep in mind that sometimes information is lost, for instance when you cast a `TIMESTAMP` to a `DATE`. Also, the fact that types are `CAST`-compatible is in itself no guarantee that a conversion will succeed. “`CAST(123456789 as SMALLINT)`” will definitely result in an error, as will “`CAST('Judgement Day' as DATE)`”.

### Casting Parameters

Since Firebird 2.0, you can cast statement parameters to a datatype:

```sql
cast (?) as integer
```

This gives you control over the type of the parameter set up by the engine. Please notice that with statement parameters, you always need a full-syntax cast — shorthand casts are not supported.

### Casting to a Domain or its Type

Firebird 2.1 and above support casting to a domain or its base type. When casting to a domain, any
constraints (NOT NULL and/or CHECK) declared for the domain must be satisfied, or the cast will fail. Please be aware that a CHECK passes if it evaluates to TRUE or NULL! So, given the following statements:

```
create domain quint as int check (value >= 5000);
select cast (2000 as quint) from rdb$database;  
select cast (8000 as quint) from rdb$database;  
select cast (null as quint) from rdb$database;  
```

only cast number 1 will result in an error.

When the TYPE OF modifier is used, the expression is cast to the base type of the domain, ignoring any constraints. With domain quint defined as above, the following two casts are equivalent and will both succeed:

```
select cast (2000 as type of quint) from rdb$database;
select cast (2000 as int) from rdb$database;
```

If TYPE OF is used with a (VAR)CHAR type, its character set and collation are retained:

```
create domain iso20 varchar(20) character set iso8859_1;
create domain dunl20 varchar(20) character set iso8859_1 collate du_nl;
create table zinnen (zin varchar(20));
commit;
insert into zinnen values ('Deze');
insert into zinnen values ('Die');
insert into zinnen values ('die');
insert into zinnen values ('deze');
select cast(zin as type of iso20) from zinnen order by 1;
-- returns Deze -> Die -> deze -> die
select cast(zin as type of dunl20) from zinnen order by 1;
-- returns deze -> Deze -> die -> Die
```

If a domain’s definition is changed, existing CASTs to that domain or its type may become invalid. If these CASTs occur in PSQL modules, their invalidation may be detected. See the note *The RDB$VALID_BLR field*, in Appendix A.

### Casting to a Column’s Type

In Firebird 2.5 and above, it is possible to cast expressions to the type of an existing table or view column. Only the type itself is used; in the case of string types, this includes the character set but not the collation. Constraints and default values of the source column are not applied.

```
create table ttt 
  s varchar(40) character set utf8 collate unicode_ci_ai
```
Commit;

```sql
select cast ('Jag har många vänner' as type of column ttt.s) from rdb$database;
```

**Warnings**

If a column's definition is altered, existing `CAST`s to that column's type may become invalid. If these `CAST`s occur in PSQL modules, their invalidation may be detected. See the note *The RDB$VALID_BLR field*, in Appendix A.

**Cast Examples**

A full-syntax cast:

```sql
select cast ('12' || '-June-' || '1959' as date) from rdb$database
```

A shorthand string-to-date cast:

```sql
update People set AgeCat = 'Old'
where BirthDate < date '1-Jan-1943'
```

Notice that you can drop even the shorthand cast from the example above, as the engine will understand from the context (comparison to a `DATE` field) how to interpret the string:

```sql
update People set AgeCat = 'Old'
where BirthDate < '1-Jan-1943'
```

But this is not always possible. The cast below cannot be dropped, otherwise the engine would find itself with an integer to be subtracted from a string:

```sql
select date 'today' - 7 from rdb$database
```

### 8.6. Bitwise Functions

#### 8.6.1. `BIN_AND()`

*Available in*

DSQL, PSQL

*Possible name conflict*

YES → Read details
Result type
SMALLINT, INTEGER or BIGINT

- SMALLINT result is returned only if all the arguments are explicit SMALLINTs or NUMERIC(n, 0) with n <= 4; otherwise small integers return an INTEGER result.

Syntax

```
BIN_AND (number, number [, number ...])
```

**Table 164. BIN_AND Function Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Any integer number (literal, smallint/integer/bigint, numeric/decimal with scale 0)</td>
</tr>
</tbody>
</table>

Returns the result of the bitwise AND operation on the argument(s).

See also

`BIN_OR()`, `BIN_XOR()`

### 8.6.2. BIN_NOT()

**Available in**
DSQL, PSQL

Result type
SMALLINT, INTEGER or BIGINT

- SMALLINT result is returned only if all the arguments are explicit SMALLINTs or NUMERIC(n, 0) with n <= 4; otherwise small integers return an INTEGER result.

Syntax

```
BIN_NOT (number)
```

**Table 165. BIN_NOT Function Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Any integer number (literal, smallint/integer/bigint, numeric/decimal with scale 0)</td>
</tr>
</tbody>
</table>

Returns the result of the bitwise NOT operation on the argument, i.e. one's complement.

See also

`BIN_OR()`, `BIN_XOR()` and others in this set.
8.6.3. BIN_OR()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
SMALLINT, INTEGER or BIGINT

- SMALLINT result is returned only if all the arguments are explicit SMALLIN\(\text{T}\)s or NUMERIC(\(n\), 0) with \(n \leq 4\); otherwise small integers return an INTEGER result.

Syntax

```
BIN_OR (number, number [, number ...])
```

Table 166. BIN_OR Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Any integer number (literal, smallint/integer/bigint, numeric/decimal with scale 0)</td>
</tr>
</tbody>
</table>

Returns the result of the bitwise OR operation on the argument(s).

See also

BIN_AND(), BIN_XOR()  

8.6.4. BIN_SHL()

Available in
DSQL, PSQL

Result type
BIGINT

Syntax

```
BIN_SHL (number, shift)
```

Table 167. BIN_SHL Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>A number of an integer type</td>
</tr>
<tr>
<td>shift</td>
<td>The number of bits the number value is shifted by</td>
</tr>
</tbody>
</table>

Returns the first argument bitwise left-shifted by the second argument, i.e. \(a \ll b\) or \(a \cdot 2^b\).
See also
BIN_SHR()

8.6.5. BIN_SHR()

Available in
DSQL, PSQL

Result type
BIGINT

Syntax

```
BIN_SHR (number, shift)
```

Table 168. BIN_SHR Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>A number of an integer type</td>
</tr>
<tr>
<td>shift</td>
<td>The number of bits the number value is shifted by</td>
</tr>
</tbody>
</table>

Returns the first argument bitwise right-shifted by the second argument, i.e. \( a >> b \) or \( a/2^b \).

- The operation performed is an arithmetic right shift (SAR), meaning that the sign of the first operand is always preserved.

See also
BIN_SHL()

8.6.6. BIN_XOR()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Result type
SMALLINT, INTEGER or BIGINT

- SMALLINT result is returned only if all the arguments are explicit SMALLINTs or NUMERIC\( (n, 0) \) with \( n <= 4 \); otherwise small integers return an INTEGER result.

Syntax

```
BIN_XOR (number, number [, number ...])
```

Table 169. BIN_XOR Function Parameters
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Any integer number (literal, smallint/integer/bigint, numeric/decimal with scale 0)</td>
</tr>
</tbody>
</table>

Returns the result of the bitwise XOR operation on the argument(s).

*See also*

`BIN_AND()`, `BIN_OR()`

## 8.7. UUID Functions

### 8.7.1. CHAR_TO_UUID()

*Available in*

DSQL, PSQL

*Result type*

CHAR(16) CHARACTER SET OCTETS

*Syntax*

```sql
CHAR_TO_UUID (ascii_uuid)
```

<table>
<thead>
<tr>
<th>Table 170. CHAR_TO_UUID Function Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>ascii_uuid</td>
</tr>
</tbody>
</table>

Converts a human-readable 36-char UUID string to the corresponding 16-byte UUID.

**CHAR_TO_UUID Examples**

```sql
select char_to_uuid('A0bF4E45-3029-2a44-D493-4998c9b439A3') from rdb$database
-- returns A0BF4E4530292A44D4934998C9B439A3 (16-byte string)

select char_to_uuid('A0bF4E45-3029-2A44-X493-4998c9b439A3') from rdb$database
-- error: -Human readable UUID argument for CHAR_TO_UUID must have hex digit at position 20 instead of "X (ASCII 88)"
```

*See also*

`UUID_TO_CHAR()`, `GEN_UUID()`
8.7.2. GEN_UUID()

Available in
DSQL, PSQL

Result type
CHAR(16) CHARACTER SET OCTETS

Syntax

GEN_UUID ()

Returns a universally unique ID as a 16-byte character string.

GEN_UUID Example

```
select gen_uuid() from rdb$database
-- returns e.g. 017347BFE212B2479C00FA4323B36320 (16-byte string)
```

See also
UUID_TO_CHAR(), CHAR_TO_UUID()

8.7.3. UUID_TO_CHAR()

Available in
DSQL, PSQL

Result type
CHAR(36)

Syntax

UUID_TO_CHAR (uuid)

Table 171. UUID_TO_CHAR Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uuid</td>
<td>16-byte UUID</td>
</tr>
</tbody>
</table>

Converts a 16-byte UUID to its 36-character, human-readable ASCII representation.

UUID_TO_CHAR Examples

```
select uuid_to_char(x'876C45F4569B320DBCB4735AC3509E5F') from rdb$database
-- returns '876C45F4-569B-320D-BCB4-735AC3509E5F'
select uuid_to_char(gen_uuid()) from rdb$database
```

429
8.8. Functions for Sequences (Generators)

8.8.1. GEN_ID()

Available in
DSQL, ESQL, PSQL

Result type
BIGINT

Syntax

\[
\text{GEN_ID (generator-name, step)}
\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generator-name</td>
<td>Name of a generator (sequence) that exists. If it has been defined in double quotes with a case-sensitive identifier, it must be used in the same form unless the name is all upper-case.</td>
</tr>
<tr>
<td>step</td>
<td>An integer expression</td>
</tr>
</tbody>
</table>

Increments a generator or sequence and returns its new value. If step equals 0, the function will leave the value of the generator unchanged and return its current value.

- From Firebird 2.0 onward, the SQL-compliant \text{NEXT VALUE FOR} syntax is preferred, except when an increment other than 1 is needed.

If the value of the step parameter is less than zero, it will decrease the value of the generator. Attention! You should be extremely cautious with such manipulations in the database, as they could compromise data integrity.

\text{GEN_ID Example}

\[
\text{new.rec_id} = \text{gen_id} (\text{gen.recnum}, 1);
\]

See also
\text{NEXT VALUE FOR, CREATE SEQUENCE (GENERATOR)}
8.9. Conditional Functions

8.9.1. COALESCE()

Available in
DSQL, PSQL

Result type
Depends on input

Syntax

```
COALESCE (<exp1>, <exp2> [, <expN> ... ])
```

Table 173. COALESCE Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp1, exp2 ... expN</td>
<td>A list of expressions of any compatible types</td>
</tr>
</tbody>
</table>

The COALESCE function takes two or more arguments and returns the value of the first non-NULL argument. If all the arguments evaluate to NULL, the result is NULL.

COALESCE Examples

This example picks the Nickname from the Persons table. If it happens to be NULL, it goes on to FirstName. If that too is NULL, "'Mr./Mrs.'" is used. Finally, it adds the family name. All in all, it tries to use the available data to compose a full name that is as informal as possible. Notice that this scheme only works if absent nicknames and first names are really NULL: if one of them is an empty string instead, COALESCE will happily return that to the caller.

```
select 
  coalesce (Nickname, FirstName, 'Mr./Mrs.') || ' ' || LastName
as Fullname
from Persons
```

See also

IIF(), NULLIF(), CASE

8.9.2. DECODE()

Available in
DSQL, PSQL

Result type
Depends on input
**Syntax**

```sql
DECODE(<testexpr>,
    <expr1>, <result1>
    [<expr2>, <result2> ...]
    [, <defaultresult>])
```

The equivalent `CASE` construct:

```sql
CASE <testexpr>
  WHEN <expr1> THEN <result1>
  [WHEN <expr2> THEN <result2> ...]
  [ELSE <defaultresult>]
END
```

**Table 174. DECODE Function Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>testexpr</td>
<td>An expression of any compatible type that is compared to the expressions expr1, expr2 ... exprN</td>
</tr>
<tr>
<td>expr1, expr2, ... exprN</td>
<td>Expressions of any compatible types, to which the testexpr expression is compared</td>
</tr>
<tr>
<td>result1, result2, ... resultN</td>
<td>Returned values of any type</td>
</tr>
<tr>
<td>defaultresult</td>
<td>The expression to be returned if none of the conditions is met</td>
</tr>
</tbody>
</table>

DECODE is a shorthand for the so-called **“simple CASE” construct**, in which a given expression is compared to a number of other expressions until a match is found. The result is determined by the value listed after the matching expression. If no match is found, the default result is returned, if present. Otherwise, NULL is returned.

Matching is done with the ‘=’ operator, so if testexpr is NULL, it won’t match any of the exprs, not even those that are NULL.

**DECODE Examples**

```sql
select name,
    age,
    decode(upper(sex),
        'M', 'Male',
        'F', 'Female',
        'Unknown'),
    religion
from people
```

*See also*
CASE, Simple CASE

8.9.3. IIF()

Available in
DSQL, PSQL

Result type
Depends on input

Syntax

\[
\text{IIF} \left( \text{<condition>}, \text{ResultT}, \text{ResultF} \right)
\]

Table 175. IIF Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>A true</td>
</tr>
<tr>
<td>resultT</td>
<td>The value returned if the condition is true</td>
</tr>
<tr>
<td>resultF</td>
<td>The value returned if the condition is false</td>
</tr>
</tbody>
</table>

IIF takes three arguments. If the first evaluates to true, the second argument is returned; otherwise the third is returned.

IIF could be likened to the ternary “?:” operator in C-like languages.

IIF(Cond, Result1, Result2) is a shorthand for “CASE WHEN <Cond> THEN Result1 ELSE Result2 END”.

IIF Examples

\[
\text{select iif( sex = 'M', 'Sir', 'Madam' ) from Customers}
\]

See also

CASE, DECODE()

8.9.4. MAXVALUE()

Available in
DSQL, PSQL

Result type
Varies according to input — result will be of the same data type as the first expression in the list (expr1).
Syntax

MAXVALUE (<expr1> [, ... , <exprN> ])

Table 176. MAXVALUE Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr1 ... exprN</td>
<td>List of expressions of compatible types</td>
</tr>
</tbody>
</table>

Returns the maximum value from a list of numerical, string, or date/time expressions. This function fully supports text BLOBs of any length and character set.

If one or more expressions resolve to NULL, MAXVALUE returns NULL. This behaviour differs from the aggregate function MAX.

MAXVALUE Examples

```
SELECT MAXVALUE(PRICE_1, PRICE_2) AS PRICE
FROM PRICELIST
```

See also

MINVALUE()

8.9.5. MINVALUE()

Available in
DSQL, PSQL

Result type

Varies according to input—result will be of the same data type as the first expression in the list (expr1).

Syntax

```
MINVALUE (<expr1> [, ... , <exprN> ])
```

Table 177. MINVALUE Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr1 ... exprN</td>
<td>List of expressions of compatible types</td>
</tr>
</tbody>
</table>

Returns the minimum value from a list of numerical, string, or date/time expressions. This function fully supports text BLOBs of any length and character set.

If one or more expressions resolve to NULL, MINVALUE returns NULL. This behaviour differs from the aggregate function MIN.
### MINVALUE Examples

```
SELECT MINVALUE(PRICE_1, PRICE_2) AS PRICE
     FROM PRICELIST
```

*See also*

**MAXVALUE()**

### 8.9.6. NULLIF()

*Available in*

DSQL, PSQL

*Result type*

Depends on input

*Syntax*

```
NULLIF (<exp1>, <exp2>)
```

#### Table 178. NULLIF Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp1</td>
<td>An expression</td>
</tr>
<tr>
<td>exp2</td>
<td>Another expression of a data type compatible with exp1</td>
</tr>
</tbody>
</table>

**NULLIF** returns the value of the first argument, unless it is equal to the second. In that case, **NULL** is returned.

**NULLIF Example**

```
select avg( nullif(Weight, -1) ) from FatPeople
```

This will return the average weight of the persons listed in FatPeople, excluding those having a weight of -1, since **AVG** skips **NULL** data. Presumably, -1 indicates “weight unknown” in this table. A plain **AVG(Weight)** would include the -1 weights, thus skewing the result.

*See also*

**COALESCE(), DECODE(), IIF(), CASE**
Chapter 9. Aggregate Functions

Aggregate functions operate on groups of records, rather than on individual records or variables. They are often used in combination with a GROUP BY clause.

The aggregate functions can also be used as window functions with the OVER () clause. See Window (Analytical) Functions for more information.

9.1. General-purpose Aggregate Functions

9.1.1. AVG()

Available in
DSQL, ESQL, PSQL

Result type
A numeric data type, the same as the data type of the argument.

Syntax

AVG ([ALL | DISTINCT] <expr>)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Expression. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF that returns a numeric data type. Aggregate functions are not allowed as expressions</td>
</tr>
</tbody>
</table>

AVG returns the average argument value in the group. NULL is ignored.

- Parameter ALL (the default) applies the aggregate function to all values.
- Parameter DISTINCT directs the AVG function to consider only one instance of each unique value, no matter how many times this value occurs.
- If the set of retrieved records is empty or contains only NULL, the result will be NULL.

AVG Examples

```
SELECT  
    dept_no,  
    AVG(salary)  
FROM employee  
GROUP BY dept_no
```

See also
SELECT
9.1.2. COUNT()

Available in
DSQL, ESQL, PSQL

Result type
BIGINT

Syntax
COUNT ([ALL | DISTINCT] <expr> | *)

Table 180. COUNT Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Expression. It may contain a table column, a constant, a variable, an</td>
</tr>
<tr>
<td></td>
<td>expression, a non-aggregate function or a UDF that returns a numeric</td>
</tr>
<tr>
<td></td>
<td>data type. Aggregate functions are not allowed as expressions</td>
</tr>
</tbody>
</table>

COUNT returns the number of non-null values in a group.

- **ALL** is the default: it simply counts all values in the set that are not NULL.
- **If DISTINCT** is specified, duplicates are excluded from the counted set.
- **If COUNT (*)** is specified instead of the expression `expr`, all rows will be counted. COUNT (*) —
  - does not accept parameters
  - cannot be used with the keyword DISTINCT
  - does not take an `expr` argument, since its context is column-unspecific by definition
  - counts each `expr` argument separately and returns the number of rows in the specified table or group without omitting duplicate rows
  - counts rows containing NULL
- **If the result set is empty or contains only NULL in the specified column(s), the returned count is zero.**

COUNT Examples

```sql
SELECT dept_no, COUNT(*) AS cnt, COUNT(DISTINCT name) AS cnt_name
FROM employee
GROUP BY dept_no
```

See also
SELECT.
9.1.3. LIST()

Available in
DSQL, PSQL

Result type
BLOB

Syntax

```
LIST ([ALL | DISTINCT] <expr> [, separator ])
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Expression. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF that returns the string data type or a BLOB. Fields of numeric and date/time types are converted to strings. Aggregate functions are not allowed as expressions.</td>
</tr>
<tr>
<td>separator</td>
<td>Optional alternative separator, a string expression. Comma is the default separator</td>
</tr>
</tbody>
</table>

LIST returns a string consisting of the non-NULL argument values in the group, separated either by a comma or by a user-supplied separator. If there are no non-NULL values (this includes the case where the group is empty), NULL is returned.

- ALL (the default) results in all non-NULL values being listed. With DISTINCT, duplicates are removed, except if expr is a BLOB.
- In Firebird 2.5 and up, the optional separator argument may be any string expression. This makes it possible to specify e.g. ascii_char(13) as a separator. (This improvement has also been backported to 2.1.4.)
- The expr and separator arguments support BLOBs of any size and character set.
- Date/time and numeric arguments are implicitly converted to strings before concatenation.
- The result is a text BLOB, except when expr is a BLOB of another subtype.
- The ordering of the list values is undefined — the order in which the strings are concatenated is determined by read order from the source set which, in tables, is not generally defined. If ordering is important, the source data can be pre-sorted using a derived table or similar.

This is a trick/workaround, and it depends on implementation details of the optimizer-execution order. This trick doesn't always work, and it is not guaranteed to work across versions.

LIST Examples

1. Retrieving the list, order undefined:
SELECT LIST (display_name, '; ') FROM GR_WORK;

2. Retrieving the list in alphabetical order, using a derived table:

```sql
SELECT LIST (display_name, '; ')
FROM (SELECT display_name
     FROM GR_WORK
     ORDER BY display_name);
```

See also

SELECT

9.1.4. MAX()

Available in

DSQL, ESQL, PSQL

Result type

Returns a result of the same data type the input expression.

Syntax

```sql
MAX ([ALL | DISTINCT] <expr>)
```

Table 182. MAX Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Expression. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

MAX returns the maximum non-NULL element in the result set.

- If the group is empty or contains only NULLs, the result is NULL.
- If the input argument is a string, the function will return the value that will be sorted last if COLLATE is used.
- This function fully supports text BLOBs of any size and character set.

The DISTINCT parameter makes no sense if used with MAX() and is implemented only for compliance with the standard.

MAX Examples

```sql
SELECT dept_no,
```
MAX(salary)
FROM employee
GROUP BY dept_no

See also
MIN(), SELECT

9.1.5. MIN()

Available in
DSQL, ESQL, PSQL

Result type
Returns a result of the same data type the input expression.

Syntax

MIN ([ALL | DISTINCT] <expr>)

Table 183. MIN Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Expression. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

MIN returns the minimum non-NULL element in the result set.

- If the group is empty or contains only NULLs, the result is NULL.
- If the input argument is a string, the function will return the value that will be sorted first if COLLATE is used.
- This function fully supports text BLOBs of any size and character set.

The DISTINCT parameter makes no sense if used with MIN() and is implemented only for compliance with the standard.

MIN Examples

```
SELECT
    dept_no,
    MIN(salary)
FROM employee
GROUP BY dept_no
```

See also
MAX(), SELECT
9.1.6. **SUM()**

*Available in*
DSQL, ESQL, PSQL

*Result type*
Depends on the input type

**Syntax**

```sql
SUM ([ALL | DISTINCT] <expr>)
```

*Table 184. SUM Function Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Numeric expression. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

**SUM** calculates and returns the sum of non-null values in the group.

- If the group is empty or contains only NULLs, the result is NULL.
- ALL is the default option — all values in the set that are not NULL are processed. If DISTINCT is specified, duplicates are removed from the set and the SUM evaluation is done afterward.

The result type of **SUM** depends on the input type:

- FLOAT, DOUBLE PRECISION → DOUBLE PRECISION
- SMALLINT, INTEGER, BIGINT → BIGINT
- DECIMAL/NUMERIC(p, n) → DECIMAL/NUMERIC(18, n)

**SUM Examples**

```sql
SELECT dept_no, SUM(salary), FROM employee GROUP BY dept_no
```

*See also*

SELECT

---

9.2. **Statistical Aggregate Functions**
**9.2.1. CORR**

*Available in*
DSQL, PSQL

*Result type*
DOUBLE PRECISION

**Syntax**

```
CORR ( <expr1>, <expr2> )
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exprN</td>
<td>Numeric expression. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The CORR function return the correlation coefficient for a pair of numerical expressions.

The function \( \text{CORR}(<\text{expr1}>, <\text{expr2}>) \) is equivalent to

```
\frac{\text{COVAR}_\text{POP}(<\text{expr1}>, <\text{expr2}>)}{\text{STDDEV}_\text{POP}(<\text{expr2}>) \times \text{STDDEV}_\text{POP}(<\text{expr1}>)}
```

This is also known as the Pearson correlation coefficient.

In a statistical sense, correlation is the degree of to which a pair of variables are linearly related. A linear relation between variables means that the value of one variable can to a certain extent predict the value of the other. The correlation coefficient represents the degree of correlation as a number ranging from -1 (high inverse correlation) to 1 (high correlation). A value of 0 corresponds to no correlation.

If the group or window is empty, or contains only NULL values, the result will be NULL.

**CORR Examples**

```sql
select
    corr(alength, aheight) AS c_corr
from measure
```

*See also*

COVAR_POP, STDDEV_POP

**9.2.2. COVAR_POP**

*Available in*
Chapter 9. Aggregate Functions

DSQL, PSQL

**Result type**
DOUBLE PRECISION

**Syntax**

\[
\text{COVAR\_POP ( <expr1>, <expr2> )}
\]

**Table 186. COVAR\_POP Function Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exprN</td>
<td>Numeric expression. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function COVAR\_POP returns the population covariance for a pair of numerical expressions.

The function COVAR\_POP(<expr1>, <expr2>) is equivalent to

\[
\frac{(SUM(<expr1> \times <expr2>) - SUM(<expr1>) \times SUM(<expr2>))}{COUNT(*)}
\]

If the group or window is empty, or contains only NULL values, the result will be NULL.

**COVAR\_POP Examples**

```sql
select
  covar_pop(alength, aheight) AS c_covar_pop
from measure
```

**See also**

COVAR\_SAMP, SUM(), COUNT()

9.2.3. **COVAR\_SAMP**

**Available in**
DSQL, PSQL

**Result type**
DOUBLE PRECISION

**Syntax**

\[
\text{COVAR\_SAMP ( <expr1>, <expr2> )}
\]

**Table 187. COVAR\_SAMP Function Parameters**


The function `COVAR_SAMP` returns the sample covariance for a pair of numerical expressions.

The function `COVAR_SAMP(expr1, expr2)` is equivalent to

\[
\frac{\text{SUM}(\text{expr1} \times \text{expr2}) - \text{SUM}(\text{expr1}) \times \text{SUM}(\text{expr2})}{\text{COUNT}()} \quad \text{if } \text{COUNT}() > 1
\]

If the group or window is empty, contains only 1 row, or contains only NULL values, the result will be `NULL`.

**COVAR_SAMP Examples**

```sql
select
covar_samp(alength, aheight) AS c_covar_samp
from measure
```

See also

`COVAR_POP`, `SUM()`, `COUNT()`

### 9.2.4. STDDEV_POP

**Available in**

DSQL, PSQL

**Result type**

DOUBLE PRECISION or NUMERIC depending on the type of `expr`

**Syntax**

`STDDEV_POP ( expr )`

**Table 188. STDDEV_POP Function Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Numeric expression. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function `STDDEV_POP` returns the population standard deviation for a group or window. NULL values are skipped.

The function `STDDEV_POP(expr)` is equivalent to
If the group or window is empty, or contains only NULL values, the result will be NULL.

**STDDEV_POP Examples**

```sql
select
department_number
    stddev_pop(salary)
from employee
group by department_number
```

*See also*

STDDEV_SAMP, VAR_POP, SQRT

### 9.2.5. STDDEV_SAMP

**Available in**

DSQL, PSQL

**Result type**

DOUBLE PRECISION or NUMERIC depending on the type of `expr`

**Syntax**

```sql
STDDEV_POP ( <expr> )
```

**Table 189. STDDEV_SAMP Function Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Numeric expression. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function STDDEV_SAMP returns the sample standard deviation for a group or window. NULL values are skipped.

The function `STDDEV_SAMP(expr)` is equivalent to

```sql
SQRT(VAR_SAMP(expr))
```

If the group or window is empty, contains only 1 row, or contains only NULL values, the result will be NULL.
STDDEV_SAMP Examples

```sql
select
department
stddev_samp(salary)
from employee
group by department
```

See also
STDDEV_POP, VAR_SAMP, SQRT

9.2.6. VAR_POP

Available in
DSQL, PSQL

Result type
DOUBLE PRECISION or NUMERIC depending on the type of expr

Syntax

```sql
VAR_POP ( <expr> )
```

Table 190. VAR_POP Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Numeric expression. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function VAR_POP returns the population variance for a group or window. NULL values are skipped.

The function VAR_POP(<expr>) is equivalent to

```sql
(SUM(<expr> * <expr>) - SUM (<expr>) * SUM (<expr>) / COUNT(<expr>))
/ COUNT (<expr>)
```

If the group or window is empty, or contains only NULL values, the result will be NULL.

VAR_POP Examples

```sql
select
department
var_pop(salary)
from employee
```
See also

VAR_SAMP, SUM(), COUNT()

9.2.7. VAR_SAMP

Available in
DSQL, PSQL

Result type
DOUBLE PRECISION or NUMERIC depending on the type of expr

Syntax

VAR_SAMP ( <expr> )

Table 191. VAR_SAMP Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Numeric expression. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function VAR_POP returns the sample variance for a group or window. NULL values are skipped.

The function VAR_SAMP(<expr>) is equivalent to

\[
\frac{(\text{SUM}(\text{expr} * \text{expr}) - \text{SUM}(\text{expr}) \ast \text{SUM}(\text{expr}) / \text{COUNT}(\text{expr}))}{\text{COUNT}(\text{expr}) - 1}
\]

If the group or window is empty, contains only 1 row, or contains only NULL values, the result will be NULL.

VAR_SAMP Examples

select
department
department

var_samp(salary)

from employee

group by department

See also

VAR_POP, SUM(), COUNT()
9.3. Linear Regression Aggregate Functions

Linear regression functions are useful for trend line continuation. The trend or regression line is usually a pattern followed by a set of values. Linear regression is useful to predict future values. To continue the regression line, you need to know the slope and the point of intersection with the y-axis. As set of linear functions can be used for calculating these values.

In the function syntax, \( y \) is interpreted as an \( x \)-dependent variable.

The linear regression aggregate functions take a pair of arguments, the dependent variable expression (\( y \)) and the independent variable expression (\( x \)), which are both numeric value expressions. Any row in which either argument evaluates to NULL is removed from the rows that qualify. If there are no rows that qualify, then the result of \texttt{REGR\_COUNT} is 0 (zero), and the other linear regression aggregate functions result in NULL.

9.3.1. \texttt{REGR\_AVGX}

\textit{Available in}

DSQL, PSQL

\textit{Result type}

DOUBLE PRECISION

\textit{Syntax}

\texttt{REGR\_AVGX ( \langle y \rangle, \langle x \rangle )}

\textit{Table 192. REGR\_AVGX Function Parameters}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>Dependent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
<tr>
<td>( x )</td>
<td>Independent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function \texttt{REGR\_AVGX} calculates the average of the independent variable (\( x \)) of the regression line.

The function \texttt{REGR\_AVGX(\langle y \rangle, \langle x \rangle)} is equivalent to

\[
\text{SUM(\langle exprX \rangle) / REGR\_COUNT(\langle y \rangle, \langle x \rangle)}
\]

\[
\langle exprX \rangle :=
\text{CASE WHEN} \langle x \rangle \text{ IS NOT NULL AND} \langle y \rangle \text{ IS NOT NULL THEN} \langle x \rangle \text{ END}
\]

\textit{See also}
9.3.2. **REGR_AVGY**

*Available in*
DSQL, PSQL

*Result type*
DOUBLE PRECISION

**Syntax**

```sql
REGR_AVGY ( <y>, <x> )
```

**Table 193. REGR_AVGY Function Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>Dependent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
<tr>
<td>x</td>
<td>Independent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function `REGR_AVGY` calculates the average of the dependent variable (y) of the regression line.

The function `REGR_AVGY(<y>, <x>)` is equivalent to

```sql
SUM(<exprY>) / REGR_COUNT(<y>, <x>)
```

*<exprY> ::=*

```
CASE WHEN <x> IS NOT NULL AND <y> IS NOT NULL THEN <y> END
```

**See also**

`REGR_AVGX, REGR_COUNT, SUM()`

9.3.3. **REGR_COUNT**

*Available in*
DSQL, PSQL

*Result type*
DOUBLE PRECISION

**Syntax**

```sql
REGR_COUNT ( <y>, <x> )
```
Table 194. REGR_COUNT Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>Dependent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
<tr>
<td>x</td>
<td>Independent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function REGR_COUNT counts the number of non-empty pairs of the regression line.

The function REGR_COUNT(y, x) is equivalent to

```sql
SUM(exprXY)
```

where

```sql
exprXY :=
CASE WHEN x IS NOT NULL AND y IS NOT NULL THEN 1 END
```

See also

SUM()

9.3.4. REGR_INTERCEPT

Available in

DSQL, PSQL

Result type

DOUBLE PRECISION

Syntax

```sql
REGR_INTERCEPT ( <y>, <x> )
```

Table 195. REGR_INTERCEPT Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>Dependent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
<tr>
<td>x</td>
<td>Independent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function REGR_INTERCEPT calculates the point of intersection of the regression line with the y-axis.
The function `REGR_INTERCEPT(<y>, <x>)` is equivalent to

\[ REGR_AVGY(<y>, <x>) - REGR_SLOPE(<y>, <x>) \times REGR_AVGX(<y>, <x>) \]

**REGR_INTERCEPT Examples**

Forecasting sales volume

```sql
with recursive years (byyear) as ( 
    select 1991
    from rdb$database
    union all
    select byyear + 1
    from years
    where byyear < 2020
  ),
  s as ( 
    select 
      extract(year from order_date) as byyear,
      sum(total_value) as total_value
    from sales
    group by 1
  ),
  regr as ( 
    select 
      regr_intercept(total_value, byyear) as intercept,
      regr_slope(total_value, byyear) as slope
    from s
  )
select 
  years.byyear as byyear,
  intercept + (slope * years.byyear) as total_value
from years 
cross join regr
```

<table>
<thead>
<tr>
<th>BYYEAR</th>
<th>TOTAL_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1991</td>
<td>118377.35</td>
</tr>
<tr>
<td>1992</td>
<td>414557.62</td>
</tr>
<tr>
<td>1993</td>
<td>710737.89</td>
</tr>
<tr>
<td>1994</td>
<td>1006918.16</td>
</tr>
<tr>
<td>1995</td>
<td>1303098.43</td>
</tr>
<tr>
<td>1996</td>
<td>1599278.69</td>
</tr>
<tr>
<td>1997</td>
<td>1895458.96</td>
</tr>
<tr>
<td>1998</td>
<td>2191639.23</td>
</tr>
<tr>
<td>1999</td>
<td>2487819.50</td>
</tr>
<tr>
<td>2000</td>
<td>2783999.77</td>
</tr>
</tbody>
</table>
9.3.5. REGR_R2

Available in
DSQL, PSQL

Result type
DOUBLE PRECISION

Syntax

REGR_R2 ( <y>, <x> )

Table 196. REGR_R2 Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>Dependent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
<tr>
<td>x</td>
<td>Independent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The REGR_R2 function calculates the coefficient of determination, or R-squared, of the regression line.

The function `REGR_R2(<y>, <x>)` is equivalent to

\[
\text{POWER(CORR(<y>, <x>), 2)}
\]

See also
CORR, POWER

9.3.6. REGR_SLOPE

Available in
DSQL, PSQL

Result type
DOUBLE PRECISION
Syntax

`REGR_SLOPE( <y>, <x> )`

Table 197. `REGR_SLOPE` Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y</code></td>
<td>Dependent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
<tr>
<td><code>x</code></td>
<td>Independent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function `REGR_SLOPE` calculates the slope of the regression line.

The function `REGR_SLOPE(<y>, <x>)` is equivalent to

```
COVAR_POP(<y>, <x>) / VAR_POP(<exprX>)
```

```
<exprX> ::= CASE WHEN <x> IS NOT NULL AND <y> IS NOT NULL THEN <x> END
```

See also

`COVAR_POP, VAR_POP`

**9.3.7. REGR_SXX**

*Available in*

DSQL, PSQL

*Result type*

DOUBLE PRECISION

**Syntax**

`REGR_SXX( <y>, <x> )`

Table 198. `REGR_SXX` Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y</code></td>
<td>Dependent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Independent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function `REGR_SXX` calculates the sum of squares of the independent expression variable ($x$).

The function `REGR_SXX(<y>, <x>)` is equivalent to

$$
\text{REGR_COUNT(<y>, <x>) * VAR_POP(<exprX>)}
$$

$<\text{exprX}> :=$

```sql
\text{CASE WHEN <x> IS NOT NULL AND <y> IS NOT NULL THEN <x> END}
```

See also

`REGR_COUNT, VAR_POP`

### 9.3.8. REGR_SXY

**Available in**

DSQL, PSQL

**Result type**

DOUBLE PRECISION

**Syntax**

```sql
REGR_SXY ( <y>, <x> )
```

#### Table 199. REGR_SXY Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>Dependent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
<tr>
<td>x</td>
<td>Independent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function `REGR_SXY` calculates the sum of products of independent variable expression ($x$) times dependent variable expression ($y$).

The function `REGR_SXY(<y>, <x>)` is equivalent to

$$
\text{REGR_COUNT(<y>, <x>) * COVAR_POP(<y>, <x>)}
$$
9.3.9. REGR_SYY

Available in
DSQL, PSQL

Result type
DOUBLE PRECISION

Syntax

REGR_SYY ( <y>, <x> )

Table 200. REGR_SYY Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>Dependent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
<tr>
<td>x</td>
<td>Independent variable of the regression line. It may contain a table column, a constant, a variable, an expression, a non-aggregate function or a UDF. Aggregate functions are not allowed as expressions.</td>
</tr>
</tbody>
</table>

The function REGR_SYY calculates the sum of squares of the dependent variable (y).

The function REGR_SYY(<y>, <x>) is equivalent to

```
REGR_COUNT(<y>, <x>) * VAR_POP(<exprY>)
<exprY> ::= CASE WHEN <x> IS NOT NULL AND <y> IS NOT NULL THEN <y> END
```

See also
REGR_COUNT, VAR_POP
Chapter 10. Window (Analytical) Functions

According to the SQL specification, window functions (also known as analytical functions) are a kind of aggregation, but one that does not “filter” the result set of a query. The rows of aggregated data are mixed with the query result set.

The window functions are used with the `OVER` clause. They may appear only in the `SELECT` list or the `ORDER BY` clause of a query.

 Besides the `OVER` clause, Firebird window functions may be partitioned and ordered.

Syntax

```
<window-function> ::=  
  <window-function-name>([, <expr> ...]) OVER <window-specification>

<window-function-name> ::=  
  <aggregate-function>  
  | <ranking-function>  
  | <navigational-function>

<ranking-function> ::=  
  RANK | DENSE_RANK | ROW_NUMBER

<navigational-function>  
  LEAD | LAG | FIRST_VALUE | LAST_VALUE | NTH_VALUE

>window-specification ::=  
  ( [ <window-partition> ] [ <window-order> ] )

>window-partition ::=  
  [PARTITION BY <expr> [, <expr> ...]]

>window-order ::=  
  [ORDER BY  
    <expr> [<direction>] [<nulls placement>]  
    [, <expr> [<direction>] [<nulls placement>] ]...]

<direction ::= {ASC | DESC}

<n nulls placement ::= NULLS {FIRST | LAST}
```

Table 201. Window Function Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Expression. May contain a table column, constant, variable, expression, scalar or aggregate function. Window functions are not allowed as an expression.</td>
</tr>
<tr>
<td>aggregate_function</td>
<td>An aggregate function used as a window function</td>
</tr>
</tbody>
</table>
10.1. Aggregate Functions as Window Functions

All aggregate functions can be used as window functions, by adding the `OVER` clause.

Imagine a table `EMPLOYEE` with columns `ID`, `NAME` and `SALARY`, and the need to show each employee with his respective salary and the percentage of his salary over the payroll.

A normal query could achieve this, as follows:

```sql
select
  id,
  department,
  salary,
  salary / (select sum(salary) from employee) portion
from employee
order by id;
```

<table>
<thead>
<tr>
<th>id</th>
<th>department</th>
<th>salary</th>
<th>portion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R &amp; D</td>
<td>10.00</td>
<td>0.2040</td>
</tr>
<tr>
<td>2</td>
<td>SALES</td>
<td>12.00</td>
<td>0.2448</td>
</tr>
<tr>
<td>3</td>
<td>SALES</td>
<td>8.00</td>
<td>0.1632</td>
</tr>
<tr>
<td>4</td>
<td>R &amp; D</td>
<td>9.00</td>
<td>0.1836</td>
</tr>
<tr>
<td>5</td>
<td>R &amp; D</td>
<td>10.00</td>
<td>0.2040</td>
</tr>
</tbody>
</table>

The query is repetitive and lengthy to run, especially if `EMPLOYEE` happens to be a complex view.

The same query could be specified in a much faster and more elegant way using a window function:

```sql
select
  id,
  department,
  salary,
  salary / sum(salary) OVER () portion
from employee
order by id;
```

Here, `sum(salary) OVER ()` is computed with the sum of all `SALARY` from the query (the `EMPLOYEE` table).

10.2. Partitioning

Like aggregate functions, that may operate alone or in relation to a group, window functions may also operate on a group, which is called a “partition”. 
Syntax

\[ \langle \text{window function} \rangle (...) \text{ OVER (PARTITION BY <expr> [, <expr> ...])} \]

Aggregation over a group could produce more than one row, so the result set generated by a partition is joined with the main query using the same expression list as the partition.

Continuing the \text{EMPLOYEE} example, instead of getting the portion of each employee's salary over the all-employees total, we would like to get the portion based on just the employees in the same department:

```sql
select
    id,
    department,
    salary,
    salary / sum(salary) \text{ OVER (PARTITION BY department)} \text{ portion}
from employee
order by id;
```

Results

<table>
<thead>
<tr>
<th>id</th>
<th>department</th>
<th>salary</th>
<th>portion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R &amp; D</td>
<td>10.00</td>
<td>0.3448</td>
</tr>
<tr>
<td>2</td>
<td>SALES</td>
<td>12.00</td>
<td>0.6000</td>
</tr>
<tr>
<td>3</td>
<td>SALES</td>
<td>8.00</td>
<td>0.4000</td>
</tr>
<tr>
<td>4</td>
<td>R &amp; D</td>
<td>9.00</td>
<td>0.3103</td>
</tr>
<tr>
<td>5</td>
<td>R &amp; D</td>
<td>10.00</td>
<td>0.3448</td>
</tr>
</tbody>
</table>

10.3. Ordering

The \text{ORDER BY} sub-clause can be used with or without partitions. The \text{ORDER BY} clause within \text{OVER} specifies the order in which the window function will process rows. This order does not have to be the same as the order rows appear in the output.

There is an important concept associated with window functions: for each row there is a set of rows in its partition called the \textit{window frame}. By default, when specifying \text{ORDER BY}, the frame consists of all lines from the beginning of the partition to the current row and rows equal to the current \text{ORDER BY} expression. Without \text{ORDER BY}, the default frame consists of all rows in the partition.

As a result, for standard aggregate functions, the \text{ORDER BY} clause produces partial aggregation results as rows are processed.

Example

```sql
select
    id,
    salary,
```
sum(salary) over (order by salary) cumul_salary
from employee
order by salary;

Results

<table>
<thead>
<tr>
<th>id</th>
<th>salary</th>
<th>cumul_salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8.00</td>
<td>8.00</td>
</tr>
<tr>
<td>4</td>
<td>9.00</td>
<td>17.00</td>
</tr>
<tr>
<td>1</td>
<td>10.00</td>
<td>37.00</td>
</tr>
<tr>
<td>5</td>
<td>10.00</td>
<td>37.00</td>
</tr>
<tr>
<td>2</td>
<td>12.00</td>
<td>49.00</td>
</tr>
</tbody>
</table>

Then cumul_salary returns the partial/accumulated (or running) aggregation (of the SUM function). It may appear strange that 37.00 is repeated for the ids 1 and 5, but that is how it should work. The ORDER BY keys are grouped together and the aggregation is computed once (but summing the two 10.00). To avoid this, you can add the ID field to the end of the ORDER BY clause.

It's possible to use multiple windows with different orders, and ORDER BY parts like ASC/DESC and NULLS FIRST/LAST.

With a partition, ORDER BY works the same way, but at each partition boundary the aggregation is reset.

All aggregation functions can use ORDER BY, except for LIST().

### 10.4. Ranking Functions

The ranking functions compute the ordinal rank of a row within the window partition.

These functions can be used with or without partitioning and ordering. However, using them without ordering almost never makes sense.

The ranking functions can be used to create different type of incremental counters. Consider SUM(1) OVER (ORDER BY SALARY) as an example of what they can do, each of them in a different way. Following is an example query, also comparing with the SUM behavior.

```
select
  id, salary,
dense_rank() over (order by salary),
rank() over (order by salary),
row_number() over (order by salary),
sum(1) over (order by salary)
from employee
order by salary;
```
The difference between DENSE_RANK and RANK is that there is a gap related to duplicate rows (relative to the window ordering) only in RANK. DENSE_RANK continues assigning sequential numbers after the duplicate salary. On the other hand, ROW_NUMBER always assigns sequential numbers, even when there are duplicate values.

### 10.4.1. DENSE_RANK

**Available in**

DSQL, PSQL

**Result type**

BIGINT

**Syntax**

\[
\text{DENSE_RANK}() \text{ OVER } \langle \text{window-specification} \rangle
\]

Returns the rank of rows in a partition of a result set without ranking gaps. Rows with the same window-order values get the same rank within the partition window-partition, if specified. The dense rank of a row is equal to the number of different rank values in the partition preceding the current row, plus one.

**DENSE_RANK Examples**

```sql
select
    id,
    salary,
    dense_rank() over (order by salary)
from employee
order by salary;
```

**Result**

<table>
<thead>
<tr>
<th>id</th>
<th>salary</th>
<th>dense_rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8.00</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>9.00</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>10.00</td>
<td>3</td>
</tr>
</tbody>
</table>

Chapter 10. Window (Analytical) Functions
10.4.2. RANK

Available in
DSQL, PSQL

Result type
BIGINT

Syntax

RANK () OVER <window-specification>

Returns the rank of each row in a partition of the result set. Rows with the same values of window-order get the same rank within the partition _window-partition, if specified. The rank of a row is equal to the number of rank values in the partition preceding the current row, plus one.

RANK Examples

```
select
  id,
  salary,
  rank() over (order by salary)
from employee
order by salary;
```

Result

```
id  salary  rank
---  ------  ----- 
3    8.00   1      
4    9.00   2      
1    10.00  3      
5    10.00  3      
2    12.00  5      
```

See also

DENSE_RANK, ROW_NUMBER

10.4.3. ROW_NUMBER

Available in
DSQL, PSQL

Result type
BIGINT

Syntax

```sql
ROW_NUMBER () OVER <window-specification>
```

Returns the sequential row number in the partition of the result set, where 1 is the first row in each of the partitions.

**ROW_NUMBER Examples**

```sql
select
    id,
    salary,
    row_number() over (order by salary)
from employee
order by salary;
```

**Result**

<table>
<thead>
<tr>
<th>id</th>
<th>salary</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8.00</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>9.00</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>10.00</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>10.00</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>12.00</td>
<td>5</td>
</tr>
</tbody>
</table>

**See also**

DENSE_RANK, RANK

### 10.5. Navigational Functions

The navigational functions get the simple (non-aggregated) value of an expression from another row of the query, within the same partition.

FIRST_VALUE, LAST_VALUE and NTH_VALUE also operate on a window frame. Currently, Firebird always applies a frame from the first to the current row of the partition, not to the last. This is equivalent to using the SQL standard syntax (currently not supported by Firebird):

```
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

This is likely to produce strange or unexpected results for NTH_VALUE and especially LAST_VALUE.
Example of Navigational Functions

```
select
    id,
    salary,
    first_value(salary) over (order by salary),
    last_value(salary) over (order by salary),
    nth_value(salary, 2) over (order by salary),
    lag(salary) over (order by salary),
    lead(salary) over (order by salary)
from employee
order by salary;
```

Results

<table>
<thead>
<tr>
<th>id</th>
<th>salary</th>
<th>first_value</th>
<th>last_value</th>
<th>nth_value</th>
<th>lag</th>
<th>lead</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8.00</td>
<td>8.00</td>
<td>8.00</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
<td>9.00</td>
</tr>
<tr>
<td>4</td>
<td>9.00</td>
<td>8.00</td>
<td>9.00</td>
<td>9.00</td>
<td>8.00</td>
<td>10.00</td>
</tr>
<tr>
<td>1</td>
<td>10.00</td>
<td>8.00</td>
<td>10.00</td>
<td>9.00</td>
<td>9.00</td>
<td>10.00</td>
</tr>
<tr>
<td>5</td>
<td>10.00</td>
<td>8.00</td>
<td>10.00</td>
<td>9.00</td>
<td>10.00</td>
<td>12.00</td>
</tr>
<tr>
<td>2</td>
<td>12.00</td>
<td>8.00</td>
<td>12.00</td>
<td>9.00</td>
<td>10.00</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

10.5.1. FIRST_VALUE

Available in
DSQL, PSQL

Result type

The same as type as expr

Syntax

```
FIRST_VALUE ( <expr> ) OVER <window-specification>
```

Table 202. Arguments of FIRST_VALUE

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Expression. May contain a table column, constant, variable, expression, scalar function. Aggregate functions are not allowed as an expression.</td>
</tr>
</tbody>
</table>

Returns the first value from the current partition.

See also

LAST_VALUE, NTH_VALUE
10.5.2. LAG

Available in
DSQL, PSQL

Result type
The same as type as expr

Syntax

```
LAG ( <expr> [, <offset [, <default>]]
OVER <window-specification>
```

Table 203. Arguments of LAG

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Expression. May contain a table column, constant, variable, expression, scalar function. Aggregate functions are not allowed as an expression.</td>
</tr>
<tr>
<td>offset</td>
<td>The offset in rows before the current row to get the value identified by expr. If offset is not specified, the default is 1. offset can be a column, subquery or other expression that results in a positive integer value, or another type that can be implicitly converted to BIGINT. offset cannot be negative (use LEAD instead).</td>
</tr>
<tr>
<td>default</td>
<td>The default value to return if offset points outside the partition. Default is NULL.</td>
</tr>
</tbody>
</table>

The LAG function provides access to the row in the current partition with a given offset before the current row. If offset points outside the current partition, default will be returned, or NULL if no default was specified.

offset can be a parameter, but explicit casting to INTEGER or BIGINT is currently required (eg LAG(somecolumn, cast(? as bigint))). See CORE-6421

LAG Examples

Suppose you have RATE table that stores the exchange rate for each day. To trace the change of the exchange rate over the past five days you can use the following query.

```
select
  bydate,
  cost,
  cost - lag(cost) over (order by bydate) as change,
  100 * (cost - lag(cost) over (order by bydate)) /
  lag(cost) over (order by bydate) as percent_change
from rate
where bydate between dateadd(-4 day to current_date)
```
and current_date
order by bydate

Result

<table>
<thead>
<tr>
<th>bydate</th>
<th>cost</th>
<th>change</th>
<th>percent_change</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.10.2014</td>
<td>31.00</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>28.10.2014</td>
<td>31.53</td>
<td>0.53</td>
<td>1.7096</td>
</tr>
<tr>
<td>29.10.2014</td>
<td>31.40</td>
<td>-0.13</td>
<td>-0.4123</td>
</tr>
<tr>
<td>30.10.2014</td>
<td>31.67</td>
<td>0.27</td>
<td>0.8598</td>
</tr>
<tr>
<td>31.10.2014</td>
<td>32.00</td>
<td>0.33</td>
<td>1.0419</td>
</tr>
</tbody>
</table>

See also

LEAD

10.5.3. LAST_VALUE

Available in
DSQL, PSQL

Result type
The same as type as expr

Syntax

LAST_VALUE ( <expr> ) OVER <window-specification>

Table 204. Arguments of LAST_VALUE

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Expression. May contain a table column, constant, variable, expression, scalar function. Aggregate functions are not allowed as an expression.</td>
</tr>
</tbody>
</table>

Returns the last value from the current partition.

See also

FIRST_VALUE, NTH_VALUE

10.5.4. LEAD

Available in
DSQL, PSQL

Result type
The same as type as expr
Syntax

LEAD ( <expr> [, <offset> [, <default>]])
OVER <window-specification>

Table 205. Arguments of LEAD

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Expression. May contain a table column, constant, variable, expression, scalar function. Aggregate functions are not allowed as an expression.</td>
</tr>
<tr>
<td>offset</td>
<td>The offset in rows after the current row to get the value identified by expr. If offset is not specified, the default is 1. offset can be a column, subquery or other expression that results in a positive integer value, or another type that can be implicitly converted to BIGINT. offset cannot be negative (use LAG instead).</td>
</tr>
<tr>
<td>default</td>
<td>The default value to return if offset points outside the partition. Default is NULL.</td>
</tr>
</tbody>
</table>

The LEAD function provides access to the row in the current partition with a given offset after the current row.

If offset points outside the current partition, default will be returned, or NULL if no default was specified.

 offssett can be a parameter, but explicit casting to INTEGER or BIGINT is currently required (eg LEAD(somecolumn, cast(? as bigint)). See CORE-6421

See also
LAG

10.5.5. NTH_VALUE

Available in
DSQL, PSQL

Result type
The same as type as expr

Syntax

NTH_VALUE ( <expr>, <offset> )
[FROM {FIRST | LAST}]
OVER <window-specification>

Table 206. Arguments of NTH_VALUE
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Expression. May contain a table column, constant, variable, expression, scalar function. Aggregate functions are not allowed as an expression.</td>
</tr>
<tr>
<td>offset</td>
<td>The offset in rows from the start (FROM FIRST) or the last (FROM LAST) to get the value identified by expr. offset can be a column, subquery or other expression that results in a positive integer value, or another type that can be implicitly converted to BIGINT. offset cannot be zero or negative.</td>
</tr>
</tbody>
</table>

The NTH_VALUE function returns the Nth value starting from the first (FROM FIRST) or the last (FROM LAST) row of the current frame, see also note on frame for navigational functions. Offset 1 with FROM FIRST is equivalent to FIRST_VALUE, and offset 1 with FROM LAST is equivalent to LAST_VALUE.

offset can be a parameter, but explicit casting to INTEGER or BIGINT is currently required (e.g., LEAD(somecolumn, cast(? as bigint)). See CORE-6421

See also

FIRST_VALUE, LAST_VALUE

10.6. Aggregate Functions Inside Window Specification

It is possible to use aggregate functions (but not window functions) inside the OVER clause. In that case, first the aggregate function is applied to determine the windows, and only then the window functions are applied on those windows.

When using aggregate functions inside OVER, all columns not used in aggregate functions must be specified in the GROUP BY clause of the SELECT.

Using an Aggregate Function in a Window Specification

```sql
select
code_employee_group,
avg(salary) as avg_salary,
rank() over (order by avg(salary)) as salary_rank
from employee
group by code_employee_group
```
Chapter 11. Context Variables

11.1. CURRENT_CONNECTION

Available in
DSQL, PSQL

Type
BIGINT

Syntax

CURRENT_CONNECTION

CURRENT_CONNECTION contains the unique identifier of the current connection.

Its value is derived from a counter on the database header page, which is incremented for each new connection. When a database is restored, this counter is reset to zero.

Examples

select current_connection from rdb$database
execute procedure P_Login(current_connection)

11.2. CURRENT_DATE

Available in
DSQL, PSQL, ESQL

Type
DATE

Syntax

CURRENT_DATE

CURRENT_DATE returns the current server date.

Within a PSQL module (procedure, trigger or executable block), the value of CURRENT_DATE will remain constant every time it is read. If multiple modules call or trigger each other, the value will remain constant throughout the duration of the outermost module. If you need a progressing value in PSQL (e.g. to measure time intervals), use 'TODAY'.
Examples

```sql
select current_date from rdb$database
-- returns e.g. 2011-10-03
```

### 11.3. CURRENT_ROLE

**Available in**
DSQL, PSQL

**Type**
VARCHAR(31)

**Syntax**

```sql
CURRENT_ROLE
```

CURRENT_ROLE is a context variable containing the role of the currently connected user. If there is no active role, CURRENT_ROLE is 'NONE'.

CURRENT_ROLE always represents a valid role or 'NONE'. If a user connects with a non-existing role, the engine silently resets it to 'NONE' without returning an error.

**Example**

```sql
if (current_role <> 'MANAGER')
    then exception only_managers_may_delete;
else
    delete from Customers where custno = :custno;
```

### 11.4. CURRENT_TIME

**Available in**
DSQL, PSQL, ESQL

**Type**
TIME

**Syntax**

```sql
CURRENT_TIME [ (<precision>) ]
```

<pre>
<pref>::= 0 | 1 | 2 | 3
</pre>

The optional precision argument is not supported in ESQL.

*Table 207. CURRENT_TIME Parameter*
CURRENT_TIME returns the current server time. The default is 0 decimals, i.e. seconds precision.

- CURRENT_TIME has a default precision of 0 decimals, where CURRENT_TIMESTAMP has a default precision of 3 decimals. As a result, CURRENT_TIMESTAMP is not the exact sum of CURRENT_DATE and CURRENT_TIME, unless you explicitly specify a precision (i.e. CURRENT_TIME(3) or CURRENT_TIMESTAMP(0)).

- Within a PSQL module (procedure, trigger or executable block), the value of CURRENT_TIME will remain constant every time it is read. If multiple modules call or trigger each other, the value will remain constant throughout the duration of the outermost module. If you need a progressing value in PSQL (e.g. to measure time intervals), use 'NOW'.

CURRENT_TIME and Firebird 4 time zone support

Firebird 4 will support time zones. As part of this support, there will be an incompatibility with the CURRENT_TIME expression.

In Firebird 4, CURRENT_TIME will return the new TIME WITH TIME ZONE type. In order for your queries to be compatible with database code of future Firebird versions, Firebird 3.0.4 introduced the LOCALTIME expression. In Firebird 3.0, LOCALTIME is a synonym for CURRENT_TIME.

In Firebird 4, LOCALTIME will continue to work as it does now (returning TIME [WITHOUT TIME ZONE]), while CURRENT_TIME will return a different data type, TIME WITH TIME ZONE.

Unless you need to be able to downgrade your database to Firebird 3.0.3 or earlier, we recommend to start using LOCALTIME instead of CURRENT_TIME.

Examples

```sql
select current_time from rdb$database
-- returns e.g. 14:20:19.0000

select current_time(2) from rdb$database
-- returns e.g. 14:20:23.1200
```

See also

CURRENT_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP

11.5. CURRENT_TIMESTAMP

Available in
DSQL, PSQL, ESQL
Type
TIMESTAMP

Syntax

\[
\text{CURRENT\_TIMESTAMP}\ [\ (<\text{precision}>)] \\
\text{precision} ::= 0 \mid 1 \mid 2 \mid 3
\]

The optional \textit{precision} argument is not supported in ESQL.

\textit{Table 208. CURRENT\_TIMESTAMP Parameter}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{precision}</td>
<td>Precision. The default value is 3. Not supported in ESQL</td>
</tr>
</tbody>
</table>

\textit{CURRENT\_TIMESTAMP} returns the current server date and time. The default is 3 decimals, i.e. milliseconds precision.

- The default precision of \textit{CURRENT\_TIME} is 0 decimals, so \textit{CURRENT\_TIMESTAMP} is not the exact sum of \textit{CURRENT\_DATE} and \textit{CURRENT\_TIME}, unless you explicitly specify a precision (i.e. \textit{CURRENT\_TIME(3)} or \textit{CURRENT\_TIMESTAMP(0)}).
- Within a PSQL module (procedure, trigger or executable block), the value of \textit{CURRENT\_TIMESTAMP} will remain constant every time it is read. If multiple modules call or trigger each other, the value will remain constant throughout the duration of the outermost module. If you need a progressing value in PSQL (e.g. to measure time intervals), use \textit{'NOW'}.

\textit{CURRENT\_TIMESTAMP and Firebird 4 time zone support}

Firebird 4 will support time zones. As part of this support, there will be an incompatibility with the \textit{CURRENT\_TIMESTAMP} expression.

In Firebird 4, \textit{CURRENT\_TIMESTAMP} will return the new \textit{TIMESTAMP WITH TIME ZONE} type. In order for your queries to be compatible with database code of future Firebird versions, Firebird 3.0.4 introduced the \textit{LOCALTIMESTAMP} expression. In Firebird 3.0, \textit{LOCALTIMESTAMP} is a synonym for \textit{CURRENT\_TIMESTAMP}.

In Firebird 4, \textit{LOCALTIMESTAMP} will continue to work as it does now (returning \textit{TIMESTAMP [WITHOUT TIME ZONE]}), while \textit{CURRENT\_TIMESTAMP} will return a different data type, \textit{TIMESTAMP WITH TIME ZONE}.

Unless you need to be able to downgrade your database to Firebird 3.0.3 or earlier, we recommend to start using \textit{LOCALTIMESTAMP} instead of \textit{CURRENT\_TIMESTAMP}.

\textit{Examples}

\begin{verbatim}
select current_timestamp from rdb$database
-- returns e.g. 2008-08-13 14:20:19.6170
\end{verbatim}
CURRENT_TIMESTAMP(2) from rdb$database
-- returns e.g. 2008-08-13 14:20:23.1200

See also
CURRENT_TIME, LOCALTIME, LOCALTIMESTAMP

11.6. CURRENT_TRANSACTION

Available in
DSQL, PSQL

Type
BIGINT

Syntax

CURRENT_TRANSACTION

CURRENT_TRANSACTION contains the unique identifier of the current transaction.

Its value is derived from a counter on the database header page, which is incremented for each new transaction. When a database is restored, this counter is reset to zero.

Examples

select current_transaction from rdb$database

New.Txn_ID = current_transaction;

11.7. CURRENT_USER

Available in
DSQL, PSQL

Type
VARCHAR(31)

Syntax

CURRENT_USER

CURRENT_USER is a context variable containing the name of the currently connected user. It is fully equivalent to USER.
Example

create trigger bi_customers for customers before insert as
begin
  New.added_by  = CURRENT_USER;
  New.purchases = 0;
end

11.8. DELETING

Available in
PSQL

Type
BOOLEAN

Syntax

DELETING

Available in triggers only, DELETING indicates if the trigger fired for a DELETE operation. Intended for use in multi-action triggers.

Example

if (deleting) then
begin
  insert into Removed_Cars (id, make, model, removed)
  values (old.id, old.make, old.model, current_timestamp);
end

11.9. GDSCODE

Available in
PSQL

Type
INTEGER

Syntax

GDSCODE

In a “WHEN … DO” error handling block, the GDSCODE context variable contains the numerical representation of the current Firebird error code. Prior to Firebird 2.0, GDSCODE was only set in WHEN GDSCODE handlers. Now it may also be non-zero in WHEN ANY, WHEN SQLCODE, WHEN SQLSTATE and WHEN EXCEPTION blocks, provided that the condition raising the error corresponds with a Firebird error
code. Outside error handlers, GDSCODE is always 0. Outside PSQL, it doesn't exist at all.

After WHEN GDSCODE, you must use symbolic names like grant_obj_notfound etc. But the GDSCODE context variable is an INTEGER. If you want to compare it against a specific error, the numeric value must be used, e.g. 335544551 for grant_obj_notfound.

Example

```sql
when gdscode grant_obj_notfound, gdscode grant_fld_notfound,
    gdscode grant_nopriv, gdscode grant_nopriv_on_base
  do
  begin
    execute procedure log_grant_error(gdscode);
  exit;
end
```

11.10. INSERTING

Available in
PSQL

Type
BOOLEAN

Syntax

```sql
INSERTING
```

Available in triggers only, INSERTING indicates if the trigger fired because of an INSERT operation. Intended for use in multi-action triggers.

Example

```sql
if (inserting or updating) then
  begin
    if (new.serial_num is null) then
      new.serial_num = gen_id(gen_serials, 1);
  end
```

11.11. LOCALTIME

Available in
DSQL, PSQL, ESQL

Type
TIME
Syntax

```
LOCALTIME [ (<precision>) ]

<precision> ::= 0 | 1 | 2 | 3
```

The optional `precision` argument is not supported in ESQL.

**Table 209. LOCALTIME Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>precision</td>
<td>Precision. The default value is 0. Not supported in ESQL</td>
</tr>
</tbody>
</table>

LOCALTIME returns the current server time. The default is 0 decimals, i.e. seconds precision.

- LOCALTIME was introduced in Firebird 3.0.4 as an alias of `CURRENT_TIME`. In Firebird 4, `CURRENT_TIME` will return a `TIME WITH TIME ZONE` instead of a `TIME [WITHOUT TIME ZONE]`, while `LOCALTIME` will continue to return `TIME [WITHOUT TIME ZONE]`. It is recommended to switch from `CURRENT_TIME` to `LOCALTIME` for forward-compatibility with Firebird 4.

- `LOCALTIME` has a default precision of 0 decimals, where `LOCALTIMESTAMP` has a default precision of 3 decimals. As a result, `LOCALTIMESTAMP` is not the exact sum of `CURRENT_DATE` and `LOCALTIME`, unless you explicitly specify a precision (i.e. `LOCALTIME(3)` or `LOCALTIMESTAMP(0)`).

- Within a PSQL module (procedure, trigger or executable block), the value of `LOCALTIME` will remain constant every time it is read. If multiple modules call or trigger each other, the value will remain constant throughout the duration of the outermost module. If you need a progressing value in PSQL (e.g. to measure time intervals), use `'NOW'`.

**Examples**

```
select localtime from rdb$database
-- returns e.g. 14:20:19.0000

select localtime(2) from rdb$database
-- returns e.g. 14:20:23.1200
```

**See also**

`CURRENT_TIME`, `LOCALTIMESTAMP`

### 11.12. LOCALTIMESTAMP

**Available in**

DSQL, PSQL, ESQL

**Type**


TIMESTAMP

Syntax

```
LOCALTIMESTAMP [ (<precision>) ]
```

<precision> ::= 0 | 1 | 2 | 3

The optional `precision` argument is not supported in ESQL.

Table 210. LOCALTIMESTAMP Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>precision</td>
<td>Precision. The default value is 3. Not supported in ESQL</td>
</tr>
</tbody>
</table>

LOCALTIMESTAMP returns the current server date and time. The default is 3 decimals, i.e. milliseconds precision.

- **LOCALTIMESTAMP** was introduced in Firebird 3.0.4 as a synonym of **CURRENT_TIMESTAMP**. In Firebird 4, **CURRENT_TIMESTAMP** will return a `TIMESTAMP WITH TIME ZONE` instead of a `TIMESTAMP [WITHOUT TIME ZONE]`, while **LOCALTIMESTAMP** will continue to return `TIMESTAMP [WITHOUT TIME ZONE]`. It is recommended to switch from **CURRENT_TIMESTAMP** to **LOCALTIMESTAMP** for forward-compatibility with Firebird 4.

- The default precision of **LOCALTIME** is 0 decimals, so **LOCALTIMESTAMP** is not the exact sum of **CURRENT_DATE** and **LOCALTIME**, unless you explicitly specify a precision (i.e. `LOCATIME(3)` or `LOCALTIMESTAMP(0)`).

- Within a PSQL module (procedure, trigger or executable block), the value of **LOCALTIMESTAMP** will remain constant every time it is read. If multiple modules call or trigger each other, the value will remain constant throughout the duration of the outermost module. If you need a progressing value in PSQL (e.g. to measure time intervals), use `'NOW'`.

Examples

```
select localtimestamp from rdb$database
-- returns e.g. 2008-08-13 14:20:19.6170

select localtimestamp(2) from rdb$database
-- returns e.g. 2008-08-13 14:20:23.1200
```

See also

**CURRENT_TIMESTAMP, LOCALTIME**

11.13. NEW

Available in
PSQL, triggers only

Type
Record type

Syntax

```
NEW.column_name
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>column_name</td>
<td>Column name to access</td>
</tr>
</tbody>
</table>

**NEW** contains the new version of a database record that has just been inserted or updated. Starting with Firebird 2.0 it is read-only in **AFTER** triggers.

In multi-action triggers—introduced in Firebird 1.5—**NEW** is always available. However, if the trigger is fired by a **DELETE**, there will be no new version of the record. In that situation, reading from **NEW** will always return **NULL**; writing to it will cause a runtime exception.

### 11.14. 'NOW'

**Available in**
DSQL, PSQL, ESQL

**Type**
CHAR(3)

'NOW' is not a variable but a string literal. It is, however, special in the sense that when you **CAST()** it to a date/time type, you will get the current date and/or time. Since Firebird 2.0 the precision is 3 decimals, i.e. milliseconds. 'NOW' is case-insensitive, and the engine ignores leading or trailing spaces when casting.

Please be advised that the shorthand expressions are evaluated immediately at parse time and stay the same as long as the statement remains prepared. Thus, even if a query is executed multiple times, the value for e.g. “timestamp ’now’” won’t change, no matter how much time passes. If you need the value to progress (i.e. be evaluated upon every call), use a full cast.

- 'NOW' always returns the actual date/time, even in PSQL modules, where **CURRENT_DATE**, **CURRENT_TIME** and **CURRENT_TIMESTAMP** return the same value throughout the duration of the outermost routine. This makes 'NOW' useful for measuring time intervals in triggers, procedures and executable blocks.

- Except in the situation mentioned above, reading **CURRENT_DATE**, **CURRENT_TIME** and **CURRENT_TIMESTAMP** is generally preferable to casting 'NOW'. Be aware though...
that CURRENT_TIME defaults to seconds precision; to get milliseconds precision, use CURRENT_TIME(3).

**Examples**

```sql
select 'Now' from rdb$database -- returns 'Now'
select cast('Now' as date) from rdb$database -- returns e.g. 2008-08-13
select cast('now' as time) from rdb$database -- returns e.g. 14:20:19.6170
select cast('NOW' as timestamp) from rdb$database -- returns e.g. 2008-08-13 14:20:19.6170
```

**Shorthand syntax** for the last three statements:

```sql
select date 'Now' from rdb$database
select time 'now' from rdb$database
select timestamp 'NOW' from rdb$database
```

### 11.15. OLD

**Available in**

PSQL, triggers only

**Type**

Record type

**Syntax**

```sql
OLD.column_name
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>column_name</td>
<td>Column name to access</td>
</tr>
</tbody>
</table>

OLD contains the existing version of a database record just before a deletion or update. Starting with Firebird 2.0 it is read-only.

In multi-action triggers—introduced in Firebird 1.5—OLD is always available. However, if the trigger is fired by an INSERT, there is obviously no pre-existing version of the record. In that situation, reading from OLD will always return NULL;
writing to it will cause a runtime exception.

11.16. **ROW_COUNT**

*Available in*
PSQL

*Type*
INTEGER

*Syntax*

```
ROW_COUNT
```

The `ROW_COUNT` context variable contains the number of rows affected by the most recent DML statement (INSERT, UPDATE, DELETE, SELECT or FETCH) in the current trigger, stored procedure or executable block.

**Behaviour with SELECT and FETCH**

- After a singleton SELECT, `ROW_COUNT` is 1 if a data row was retrieved and 0 otherwise.
- In a FOR SELECT loop, `ROW_COUNT` is incremented with every iteration (starting at 0 before the first).
- After a FETCH from a cursor, `ROW_COUNT` is 1 if a data row was retrieved and 0 otherwise. Fetching more records from the same cursor does not increment `ROW_COUNT` beyond 1.
- In Firebird 1.5.x, `ROW_COUNT` is 0 after any type of SELECT statement.

Row_COUNT cannot be used to determine the number of rows affected by an EXECUTE STATEMENT or EXECUTE PROCEDURE command.

**Example**

```sql
update Figures set Number = 0 where id = :id;
if (row_count = 0) then
    insert into Figures (id, Number) values (:id, 0);
```

11.17. **SQLCODE**

*Available in*
PSQL

*Deprecated in*
2.5.1

*Type*
INTEGER
In a "WHEN ... DO" error handling block, the SQLCODE context variable contains the current SQL error code. Prior to Firebird 2.0, SQLCODE was only set in WHEN SQLCODE and WHEN ANY handlers. Now it may also be non-zero in WHEN GDSCODE, WHEN SQLSTATE and WHEN EXCEPTION blocks, provided that the condition raising the error corresponds with an SQL error code. Outside error handlers, SQLCODE is always 0. Outside PSQL, it doesn't exist at all.

SQLCODE is now deprecated in favour of the SQL-2003-compliant SQLSTATE status code. Support for SQLCODE and WHEN SQLCODE will be discontinued in some future version of Firebird.

Example

```sql
when any
do
begin
  if (sqlcode <> 0) then
    Msg = 'An SQL error occurred!';
  else
    Msg = 'Something bad happened!';
exception ex_custom Msg;
end
```

11.18. SQLSTATE

Available in
PSQL

Added in
2.5.1

Type
CHAR(5)

Syntax

```sql
SQLSTATE
```

In a "WHEN ... DO" error handler, the SQLSTATE context variable contains the 5-character, SQL-2003-compliant status code resulting from the statement that raised the error. Outside error handlers, SQLSTATE is always '00000'. Outside PSQL, it is not available at all.

- SQLSTATE is destined to replace SQLCODE. The latter is now deprecated in Firebird and will disappear in some future version.
• Firebird does not (yet) support the syntax “WHEN SQLSTATE … DO”. You have to use WHEN ANY and test the SQLSTATE variable within the handler.

• Each SQLSTATE code is the concatenation of a 2-character class and a 3-character subclass. Classes 00 (successful completion), 01 (warning) and 02 (no data) represent completion conditions. Every status code outside these classes is an exception. Because classes 00, 01 and 02 don't raise an error, they won't ever show up in the SQLSTATE variable.

• For a complete listing of SQLSTATE codes, consult the SQLSTATE Codes and Message Texts section in Appendix B: Exception Handling, Codes and Messages.

Example

```sql
when any do begin
  Msg = case sqlstate
    when '22003' then 'Numeric value out of range.'
    when '22012' then 'Division by zero.'
    when '23000' then 'Integrity constraint violation.'
    else 'Something bad happened! SQLSTATE = ' || sqlstate end;
  exception ex_custom Msg;
end
```

11.19. 'TODAY'

Available in

DSQL, PSQL, ESQL

Type

CHAR(5)

'TODAY' is not a variable, but a string literal. It is, however, special in the sense that when you CAST() it to a date/time type, you will get the current date. 'TODAY' is case-insensitive, and the engine ignores leading or trailing spaces when casting.

• 'TODAY' always returns the actual date, even in PSQL modules, where CURRENT_DATE, CURRENT_TIME and CURRENT_TIMESTAMP return the same value throughout the duration of the outermost routine. This makes 'TODAY' useful for measuring time intervals in triggers, procedures and executable blocks (at least if your procedures are running for days).

• Except in the situation mentioned above, reading CURRENT_DATE, is generally preferable to casting 'NOW'.

Examples

```sql
select 'Today' from rdb$database
```
-- returns 'Today'

select cast('Today' as date) from rdb$database
-- returns e.g. 2011-10-03

select cast('TODAY' as timestamp) from rdb$database
-- returns e.g. 2011-10-03 00:00:00.0000

Shorthand syntax for the last two statements:

```sql
select date 'Today' from rdb$database;
select timestamp 'TODAY' from rdb$database;
```

### 11.20. 'TOMORROW'

*Available in*

DSQL, PSQL, ESQL

*Type*

CHAR(8)

'TOMORROW' is not a variable, but a string literal. It is, however, special in the sense that when you \( \text{CAST()} \) it to a date/time type, you will get the date of the next day. See also 'TODAY'.

*Examples*

```sql
select 'Tomorrow' from rdb$database
-- returns 'Tomorrow'

select cast('Tomorrow' as date) from rdb$database
-- returns e.g. 2011-10-04

select cast('TOMORROW' as timestamp) from rdb$database
-- returns e.g. 2011-10-04 00:00:00.0000
```

Shorthand syntax for the last two statements:

```sql
select date 'Tomorrow' from rdb$database;
select timestamp 'TOMORROW' from rdb$database;
```

### 11.21. UPDATING

*Available in*

PSQL

*Type*
**BOOLEAN**

**Syntax**

```plaintext
UPDATING
```

Available in triggers only, **UPDATING** indicates if the trigger fired because of an **UPDATE** operation. Intended for use in **multi-action triggers**.

**Example**

```plaintext
if (inserting or updating) then
begin
  if (new.serial_num is null) then
    new.serial_num = gen_id(gen_serials, 1);
end
```

---

**11.22. 'YESTERDAY'**

**Available in**

DSQL, PSQL, ESQL

**Type**

CHAR(9)

'**YESTERDAY**' is not a variable, but a string literal. It is, however, special in the sense that when you `CAST()` it to a date/time type, you will get the date of the day before. See also '**TODAY**'.

**Examples**

```plaintext
select 'Yesterday' from rdb$database  
  -- returns 'Yesterday'

select cast('Yesterday as date) from rdb$database   
  -- returns e.g. 2011-10-02

select cast('YESTERDAY' as timestamp) from rdb$database   
  -- returns e.g. 2011-10-02 00:00:00.0000
```

**Shorthand syntax** for the last two statements:

```plaintext
select date 'Yesterday' from rdb$database;
select timestamp 'YESTERDAY' from rdb$database;
```

---

**11.23. USER**

**Available in**
USER is a context variable containing the name of the currently connected user. It is fully equivalent to `CURRENT_USER`.

**Example**

```sql
create trigger bi_customers for customers before insert as
begin
    New.added_by  = USER;
    New.purchases = 0;
end
```
Chapter 12. Transaction Control

Everything in Firebird happens in transactions. Units of work are isolated between a start point and end point. Changes to data remain reversible until the moment the client application instructs the server to commit them.

12.1. Transaction Statements

Firebird has a small lexicon of SQL statements that are used by client applications to start, manage, commit and reverse (roll back) the transactions that form the boundaries of all database tasks:

**SET TRANSACTION**
for configuring and starting a transaction

**COMMIT**
to signal the end of a unit of work and write changes permanently to the database

**ROLLBACK**
to reverse the changes performed in the transaction

**SAVEPOINT**
to mark a position in the log of work done, in case a partial rollback is needed

**RELEASE SAVEPOINT**
to erase a savepoint

### 12.1.1. SET TRANSACTION

*Used for*
Configuring and starting a transaction

*Available in*
DSQL, ESQL

*Syntax*

```
SET TRANSACTION
   [NAME tr_name]
   [<tr_option> ...]

<tr_option> ::= 
   READ {ONLY | WRITE} 
   | [NO] WAIT 
   | [ISOLATION LEVEL] 
   { SNAPSHOT [TABLE [STABILITY]] 
     | READ {UNCOMMITTED | COMMITTED } 
     | [NO] RECORD_VERSION } 
   | NO AUTO UNDO
```
Chapter 12. Transaction Control

The `SET TRANSACTION` statement configures the transaction and starts it. As a rule, only client applications start transactions. The exceptions are the occasions when the server starts an autonomous transaction or transactions for certain background system threads/processes, such as sweeping.

A client application can start any number of concurrently running transactions. A single connection can have multiple concurrent active transactions (though not all drivers or access components support this). A limit does exist, for the total number of running transactions in all client applications working with one particular database from the moment the database was restored from its backup copy or from the moment the database was created originally. The limit is $2^{48} = 281,474,976,710,656$ — in Firebird 3, and $2^{31}-1 = 2,147,483,647$ — in earlier versions.

All clauses in the `SET TRANSACTION` statement are optional. If the statement starting a transaction has no clauses specified in it, it the transaction will be started with default values for access mode, lock resolution mode and isolation level, which are:

```
SET TRANSACTION
```
Database drivers or access components may use different defaults. Check their documentation for details.

The server assigns integer numbers to transactions sequentially. Whenever a client starts any transaction, either explicitly defined or by default, the server sends the transaction ID to the client. This number can be retrieved in SQL using the context variable `CURRENT_TRANSACTION`.

Some database drivers—or their governing specifications—require that you configure and start transaction through API methods. In that case, using `SET TRANSACTION` is either not supported, or may result in unspecified behaviour. An example of this is JDBC and the Firebird JDBC driver Jaybird.

Check the documentation of your driver for details.

The `NAME` and `USING` clauses are only valid in ESQL.

**Transaction Name**

The optional `NAME` attribute defines the name of a transaction. Use of this attribute is available only in Embedded SQL. In ESQL applications, named transactions make it possible to have several transactions active simultaneously in one application. If named transactions are used, a host-language variable with the same name must be declared and initialized for each named transaction. This is a limitation that prevents dynamic specification of transaction names and thus, rules out transaction naming in DSQL.

**Transaction Parameters**

The main parameters of a transaction are:

- data access mode (`READ WRITE`, `READ ONLY`)
- lock resolution mode (`WAIT`, `NO WAIT`) with an optional `LOCK TIMEOUT` specification
- isolation level (`READ COMMITTED`, `SNAPSHOT`, `SNAPSHOT TABLE STABILITY`).

The `READ UNCOMMITTED` isolation level is a synonym for `READ COMMITTED`, and provided only for syntax compatibility. It provides the exact same semantics as `READ COMMITTED`, and does not allow you to view uncommitted changes of other transactions.

- a mechanism for reserving or releasing tables (the `RESERVING` clause)

**Access Mode**

The two database access modes for transactions are `READ WRITE` and `READ ONLY`. 
• If the access mode is **READ WRITE**, operations in the context of this transaction can be both read operations and data update operations. This is the default mode.

• If the access mode is **READ ONLY**, only **SELECT** operations can be executed in the context of this transaction. Any attempt to change data in the context of such a transaction will result in database exceptions. However, this does not apply to global temporary tables (GTT), which are allowed to be changed in **READ ONLY** transactions, see *Global Temporary Tables (GTT)* in Chapter *Data Definition (DDL) Statements* for details.

### Lock Resolution Mode

When several client processes work with the same database, locks may occur when one process makes uncommitted changes in a table row, or deletes a row, and another process tries to update or delete the same row. Such locks are called *update conflicts*.

Locks may occur in other situations when multiple transaction isolation levels are used.

The two lock resolution modes are **WAIT** and **NO WAIT**.

**WAIT Mode**

In the **WAIT** mode (the default mode), if a conflict occurs between two parallel processes executing concurrent data updates in the same database, a **WAIT** transaction will wait till the other transaction has finished — by committing (**COMMIT**) or rolling back (**ROLLBACK**). The client application with the **WAIT** transaction will be put on hold until the conflict is resolved.

If a **LOCK TIMEOUT** is specified for the **WAIT** transaction, waiting will continue only for the number of seconds specified in this clause. If the lock is unresolved at the end of the specified interval, the error message “Lock time-out on wait transaction” is returned to the client.

Lock resolution behaviour can vary a little, depending on the transaction isolation level.

**NO WAIT Mode**

In the **NO WAIT** mode, a transaction will immediately throw a database exception if a conflict occurs.

---

**LOCK TIMEOUT** is a separate transaction option, but can only be used for **WAIT** transactions. Specifying **LOCK TIMEOUT** with a **NO WAIT** transaction will raise an error “invalid parameter in transaction parameter block -Option *isc_tpb_lock_timeout is not valid if *isc_tpb_nowait was used previously in TPB”

### Isolation Level

Keeping the work of one database task separated from others is what isolation is about. Changes made by one statement become visible to all remaining statements executing within the same transaction, regardless of its isolation level. Changes that are in progress within other transactions remain invisible to the current transaction as long as they remain uncommitted. The isolation level and, sometimes, other attributes, determine how transactions will interact when another transaction wants to commit work.

The **ISOLATION LEVEL** attribute defines the isolation level for the transaction being started. It is the
most significant transaction parameter for determining its behavior towards other concurrently running transactions.

The three isolation levels supported in Firebird are:

- **SNAPSHOT**
- **SNAPSHOT TABLE STABILITY**
- **READ COMMITTED** with two specifications (NO RECORD_VERSION and RECORD_VERSION)

### SNAPSHOT Isolation Level

SNAPSHOT isolation level — the default level — allows the transaction to see only those changes that were committed before it was started. Any committed changes made by concurrent transactions will not be seen in a SNAPSHOT transaction while it is active. The changes will become visible to a new transaction once the current transaction is either committed or rolled back completely, but not if it is just rolled back to a savepoint.

The SNAPSHOT isolation level is also known as “concurrency”.

#### Autonomous Transactions

Changes made by autonomous transactions are not seen in the context of the SNAPSHOT transaction that launched it.

### SNAPSHOT TABLE STABILITY Isolation Level

The SNAPSHOT TABLE STABILITY — or SNAPSHOT TABLE — isolation level is the most restrictive. As in SNAPSHOT, a transaction in SNAPSHOT TABLE STABILITY isolation sees only those changes that were committed before the current transaction was started. After a SNAPSHOT TABLE STABILITY is started, no other transactions can make any changes to any table in the database that has changes pending for this transaction. Other transactions are able to read other data, but any attempt at inserting, updating or deleting by a parallel process will cause conflict exceptions.

The RESERVING clause can be used to allow other transactions to change data in some tables.

If any other transaction has an uncommitted change pending in any (non-SHARED) table listed in the RESERVING clause, trying to start a SNAPSHOT TABLE STABILITY transaction will result in an indefinite wait (default or explicit WAIT), or an exception (NO WAIT or after expiration of the LOCK TIMEOUT).

The SNAPSHOT TABLE STABILITY isolation level is also known as “consistency”.

### READ COMMITTED Isolation Level

The READ COMMITTED isolation level allows all data changes that other transactions have committed since it started to be seen immediately by the uncommitted current transaction. Uncommitted changes are not visible to a READ COMMITTED transaction.

To retrieve the updated list of rows in the table you are interested in — “refresh” — the SELECT statement just needs to be requested again, whilst still in the uncommitted READ COMMITTED transaction.
One of two modifying parameters can be specified for READ COMMITTED transactions, depending on the kind of conflict resolution desired: RECORD_VERSION and NO RECORD_VERSION. As the names suggest, they are mutually exclusive.

- **NO RECORD_VERSION** (the default value) is a kind of two-phase locking mechanism: it will make the transaction unable to write to any row that has an update pending from another transaction.
  - if **WAIT** is the lock resolution strategy specified, it will throw a lock conflict error immediately
  - with **WAIT** specified, it will wait until the other transaction either commits or is rolled back. If the other transaction is rolled back, or if it is committed and its transaction ID is older than the current transaction's ID, then the current transaction's change is allowed. A lock conflict error is returned if the other transaction was committed and its ID was newer than that of the current transaction.

- With **RECORD_VERSION** specified, the transaction reads the latest committed version of the row, regardless of other pending versions of the row. The lock resolution strategy (**WAIT** or **NO WAIT**) does not affect the behavior of the transaction at its start in any way.

**NO AUTO UNDO**

The **NO AUTO UNDO** option affects the handling of record versions (garbage) produced by the transaction in the event of rollback. With **NO AUTO UNDO** flagged, the ROLLBACK statement just marks the transaction as rolled back without deleting the record versions created in the transaction. They are left to be mopped up later by garbage collection.

**NO AUTO UNDO** might be useful when a lot of separate statements are executed that change data in conditions where the transaction is likely to be committed successfully most of the time.

The **NO AUTO UNDO** option is ignored for transactions where no changes are made.

**RESTART REQUESTS**

According to the Firebird sources, this will

> Restart all requests in the current attachment to utilize the passed transaction.

--- src/jrd/tra.cpp

The exact semantics and effects of this clause are not clear, and we recommend you do not use this clause.

**IGNORE LIMBO**

This flag is used to signal that records created by limbo transactions are to be ignored. Transactions are left “in limbo” if the second stage of a two-phase commit fails.

**Historical Note**

**IGNORE LIMBO** surfaces the TPB parameter `isc_tpb_ignore_limbo`, available in the API.
since InterBase times and is mainly used by gfix.

**RESERVING**

The **RESERVING** clause in the `SET TRANSACTION` statement reserves tables specified in the table list. Reserving a table prevents other transactions from making changes in them or even, with the inclusion of certain parameters, from reading data from them while this transaction is running.

A **RESERVING** clause can also be used to specify a list of tables that can be changed by other transactions, even if the transaction is started with the **SNAPSHOT TABLE STABILITY** isolation level.

One **RESERVING** clause is used to specify as many reserved tables as required.

**Options for RESERVING Clause**

If one of the keywords **SHARED** or **PROTECTED** is omitted, **SHARED** is assumed. If the whole **FOR** clause is omitted, **FOR SHARED READ** is assumed. The names and compatibility of the four access options for reserving tables are not obvious.

<table>
<thead>
<tr>
<th>Table 214. Compatibility of Access Options for RESERVING</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHARE</td>
</tr>
<tr>
<td>SHARE</td>
</tr>
<tr>
<td>SHARE</td>
</tr>
<tr>
<td>SHARE</td>
</tr>
<tr>
<td>SHARE</td>
</tr>
<tr>
<td>SHARE</td>
</tr>
</tbody>
</table>

The combinations of these **RESERVING** clause flags for concurrent access depend on the isolation levels of the concurrent transactions:

- **SNAPSHOT** isolation
  - Concurrent **SNAPSHOT** transactions with **SHARE** **READ** do not affect one other’s access
  - A concurrent mix of **SNAPSHOT** and **READ COMMITTED** transactions with **SHARE** **WRITE** do not affect one another’s access, but they block transactions with **SNAPSHOT TABLE STABILITY** isolation from either reading from or writing to the specified table(s)
  - Concurrent transactions with any isolation level and **PROTECTED READ** can only read data from the reserved tables. Any attempt to write to them will cause an exception
  - With **PROTECTED WRITE**, concurrent transactions with **SNAPSHOT** and **READ COMMITTED** isolation cannot write to the specified tables. Transactions with **SNAPSHOT TABLE STABILITY** isolation cannot read from or write to the reserved tables at all.

- **SNAPSHOT TABLE STABILITY** isolation
  - All concurrent transactions with **SHARE** **READ**, regardless of their isolation levels, can read from or write (if in **READ WRITE** mode) to the reserved tables
  - Concurrent transactions with **SNAPSHOT** and **READ COMMITTED** isolation levels and **SHARE** **WRITE**
can read data from and write (if in READ WRITE mode) to the specified tables but concurrent access to those tables from transactions with SNAPSHOT TABLE STABILITY is blocked completely whilst these transactions are active

- Concurrent transactions with any isolation level and PROTECTED READ can only read from the reserved tables
- With PROTECTED WRITE, concurrent SNAPSHOT and READ COMMITTED transactions can read from but not write to the reserved tables. Access by transactions with the SNAPSHOT TABLE STABILITY isolation level is blocked completely.

- **READ COMMITTED** isolation
  - With SHARED READ, all concurrent transactions with any isolation level can both read from and write (if in READ WRITE mode) to the reserved tables
  - SHARED WRITE allows all transactions in SNAPSHOT and READ COMMITTED isolation to read from and write (if in READ WRITE mode) to the specified tables and blocks access completely from transactions with SNAPSHOT TABLE STABILITY isolation
  - With PROTECTED READ, concurrent transactions with any isolation level can only read from the reserved tables
  - With PROTECTED WRITE, concurrent transactions in SNAPSHOT and READ COMMITTED isolation can read from but not write to the specified tables. Access from transactions in SNAPSHOT TABLE STABILITY isolation is blocked completely.

In Embedded SQL, the **USING** clause can be used to conserve system resources by limiting the number of databases a transaction can access. **USING** is mutually exclusive with **RESERVING**. A **USING** clause in **SET TRANSACTION** syntax is not supported in DSQL.

See also

**COMMIT**, **ROLLBACK**

### 12.1.2. COMMIT

**Used for**

Committing a transaction

**Available in**

DSQL, ESQL

**Syntax**

```sql
COMMIT [TRANSACTION tr_name] [WORK]
[RETAIN [SNAPSHOT] | RELEASE];
```

**Table 215. COMMIT Statement Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tr_name</td>
<td>Transaction name. Available only in ESQL</td>
</tr>
</tbody>
</table>
The COMMIT statement commits all work carried out in the context of this transaction (inserts, updates, deletes, selects, execution of procedures). New record versions become available to other transactions and, unless the RETAIN clause is employed, all server resources allocated to its work are released.

If any conflicts or other errors occur in the database during the process of committing the transaction, the transaction is not committed, and the reasons are passed back to the user application for handling, and the opportunity to attempt another commit or to roll the transaction back.

The TRANSACTION and RELEASE clauses are only valid in ESQL.

**COMMIT Options**

- The optional TRANSACTION `tr_name` clause, available only in Embedded SQL, specifies the name of the transaction to be committed. With no TRANSACTION clause, COMMIT is applied to the default transaction.

  In ESQL applications, named transactions make it possible to have several transactions active simultaneously in one application. If named transactions are used, a host-language variable with the same name must be declared and initialized for each named transaction. This is a limitation that prevents dynamic specification of transaction names and thus, rules out transaction naming in DSQL.

- The optional keyword WORK is supported just for compatibility with other relational database management systems that require it.

- The keyword RELEASE is available only in Embedded SQL and enables disconnection from all databases after the transaction is committed. RELEASE is retained in Firebird only for compatibility with legacy versions of InterBase. It has been superseded in ESQL by the DISCONNECT statement.

- The RETAIN [SNAPSHOT] clause is used for the “soft” commit, variously referred to amongst host languages and their practitioners as COMMIT WITH RETAIN, “CommitRetaining”, “warm commit”, et al. The transaction is committed, but some server resources are retained and a new transaction is restarted transparently with the same Transaction ID. The state of row caches and cursors is kept as it was before the soft commit.

  For soft-committed transactions whose isolation level is SNAPSHOT or SNAPSHOT TABLE STABILITY, the view of database state is not updated to reflect changes by other transactions, and the user of the application instance continues to have the same view as when the transaction started originally. Changes made during the life of the retained transaction are visible to that transaction, of course.

**Recommendation**

Use of the COMMIT statement in preference to ROLLBACK is recommended for ending transactions that only read data from the database, because COMMIT consumes fewer server resources and helps to optimize the performance of subsequent transactions.
transactions.

See also
SET TRANSACTION, ROLLBACK

12.1.3. ROLLBACK

Used for
Rolling back a transaction

Available in
DSQL, ESQL

Syntax

ROLLBACK [TRANSACTION tr_name] [WORK] [RETAIN [SNAPSHOT] | RELEASE] | ROLLBACK [WORK] TO [SAVEPOINT] sp_name

Table 216. ROLLBACK Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tr_name</td>
<td>Transaction name. Available only in ESQL</td>
</tr>
<tr>
<td>sp_name</td>
<td>Savepoint name. Available only in DSQL</td>
</tr>
</tbody>
</table>

The ROLLBACK statement rolls back all work carried out in the context of this transaction (inserts, updates, deletes, selects, execution of procedures). ROLLBACK never fails and, thus, never causes exceptions. Unless the RETAIN clause is employed, all server resources allocated to the work of the transaction are released.

The TRANSACTION and RELEASE clauses are only valid in ESQL. The ROLLBACK TO SAVEPOINT statement is not available in ESQL.

ROLLBACK Options

- The optional TRANSACTION tr_name clause, available only in Embedded SQL, specifies the name of the transaction to be committed. With no TRANSACTION clause, ROLLBACK is applied to the default transaction.

In ESQL applications, named transactions make it possible to have several transactions active simultaneously in one application. If named transactions are used, a host-language variable with the same name must be declared and initialized for each named transaction. This is a limitation that prevents dynamic specification of transaction names and thus, rules out transaction naming in DSQL.

- The optional keyword WORK is supported just for compatibility with other relational database management systems that require it.
The keyword RETAIN keyword specifies that, although all work of the transaction is to be rolled back, the transaction context is to be retained. Some server resources are retained, and the transaction is restarted transparently with the same Transaction ID. The state of row caches and cursors is kept as it was before the “soft” rollback.

For transactions whose isolation level is SNAPSHOT or SNAPSHOT TABLE STABILITY, the view of database state is not updated by the soft rollback to reflect changes by other transactions. The user of the application instance continues to have the same view as when the transaction started originally. Changes that were made and soft-committed during the life of the retained transaction are visible to that transaction, of course.

See also

SET TRANSACTION, COMMIT

ROLLBACK TO SAVEPOINT

The alternative ROLLBACK TO SAVEPOINT statement specifies the name of a savepoint to which changes are to be rolled back. The effect is to roll back all changes made within the transaction, from the specified savepoint forward until the point when ROLLBACK TO SAVEPOINT is requested.

ROLLBACK TO SAVEPOINT performs the following operations:

- Any database mutations performed since the savepoint was created are undone. User variables set with RDB$SET_CONTEXT() remain unchanged.
- Any savepoints that were created after the one named are destroyed. Savepoints earlier than the one named are preserved, along with the named savepoint itself. Repeated rollbacks to the same savepoint are thus allowed.
- All implicit and explicit record locks that were acquired since the savepoint are released. Other transactions that have requested access to rows locked after the savepoint must continue to wait until the transaction is committed or rolled back. Other transactions that have not already requested the rows can request and access the unlocked rows immediately.

See also

SAVEPOINT, RELEASE SAVEPOINT

12.1.4. SAVEPOINT

Used for
Creating a savepoint

Available in
DSQL

Syntax

```
SAVEPOINT sp_name
```

Table 217. SAVEPOINT Statement Parameter
<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>sp_name</td>
<td>Savepoint name. Available only in DSQL</td>
</tr>
</tbody>
</table>

The `SAVEPOINT` statement creates an SQL:99-compliant savepoint that acts as a marker in the “stack” of data activities within a transaction. Subsequently, the tasks performed in the “stack” can be undone back to this savepoint, leaving the earlier work and older savepoints untouched. Savepoint mechanisms are sometimes characterised as “nested transactions”.

If a savepoint already exists with the same name as the name supplied for the new one, the existing savepoint is released, and a new one is created using the supplied name.

To roll changes back to the savepoint, the statement `ROLLBACK TO SAVEPOINT` is used.

### Memory Considerations

The internal mechanism beneath savepoints can consume large amounts of memory, especially if the same rows receive multiple updates in one transaction. When a savepoint is no longer needed, but the transaction still has work to do, a `RELEASE SAVEPOINT` statement will erase it and thus free the resources.

**Sample DSQL session with savepoints**

```
CREATE TABLE TEST (ID INTEGER);
COMMIT;
INSERT INTO TEST VALUES (1);
COMMIT;
INSERT INTO TEST VALUES (2);
SAVEPOINT Y;
DELETE FROM TEST;
SELECT * FROM TEST; -- returns no rows
ROLLBACK TO Y;
SELECT * FROM TEST; -- returns two rows
ROLLBACK;
SELECT * FROM TEST; -- returns one row
```

**See also**

`ROLLBACK TO SAVEPOINT, RELEASE SAVEPOINT`

### 12.1.5. RELEASE SAVEPOINT

**Used for**

Erasing a savepoint

**Available in**

DSQL
Syntax

```
RELEASE SAVEPOINT sp_name [ONLY]
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp_name</td>
<td>Savepoint name. Available only in DSQL</td>
</tr>
</tbody>
</table>

The statement `RELEASE SAVEPOINT` erases a named savepoint, freeing up all the resources it encompasses. By default, all the savepoints created after the named savepoint are released as well. The qualifier `ONLY` directs the engine to release only the named savepoint.

See also

SAVEPOINT

12.1.6. Internal Savepoints

By default, the engine uses an automatic transaction-level system savepoint to perform transaction rollback. When a `ROLLBACK` statement is issued, all changes performed in this transaction are backed out via a transaction-level savepoint, and the transaction is then committed. This logic reduces the amount of garbage collection caused by rolled back transactions.

When the volume of changes performed under a transaction-level savepoint is getting large (~50000 records affected), the engine releases the transaction-level savepoint and uses the Transaction Inventory Page (TIP) as a mechanism to roll back the transaction if needed.

If you expect the volume of changes in your transaction to be large, you can specify the `NO AUTO UNDO` option in your `SET TRANSACTION` statement to block the creation of the transaction-level savepoint. Using the API instead, you would set the `TPB` flag `isc_tpb_no_auto_undo`.

12.1.7. Savepoints and PSQL

Transaction control statements are not allowed in PSQL, as that would break the atomicity of the statement that calls the procedure. However, Firebird does support the raising and handling of exceptions in PSQL, so that actions performed in stored procedures and triggers can be selectively undone without the entire procedure failing.

Internally, automatic savepoints are used to:

- undo all actions in the `BEGIN...END` block where an exception occurs
- undo all actions performed by the procedure or trigger or, in a selectable procedure, all actions performed since the last `SUSPEND`, when execution terminates prematurely because of an uncaught error or exception

Each PSQL exception handling block is also bounded by automatic system savepoints.
A BEGIN⋯END block does not itself create an automatic savepoint. A savepoint is created only in blocks that contain the WHEN statement for handling exceptions.
Chapter 13. Security

Databases must be secure and so must the data stored in them. Firebird provides three levels of data security: user authentication at the server level, SQL privileges within databases, and—optionally—database encryption. This chapter describes how to manage security at these three levels.

There is also a fourth level of data security: wire protocol encryption, which encrypts data in transit between client and server. Wire protocol encryption is out of scope for this Language Reference.

13.1. User Authentication

The security of the entire database depends on identifying a user and verifying its authority, a procedure known as authentication. User authentication can be performed in several ways, depending on the setting of the AuthServer parameter in the firebird.conf configuration file. This parameter contains the list of authentication plugins that can be used when connecting to the server. If the first plugin fails when authenticating, then the client can proceed with the next plugin, etc. When no plugin could authenticate the user, the user receives an error message.

The information about users authorised to access a specific Firebird server is stored in a special security database named security3.fdb. Each record in security3.fdb is a user account for one user. For each database, the security database can be overridden in the databases.conf file (parameter SecurityDatabase). Any database can be a security database, even for that database itself.

A username, consisting of up to 31 characters, is an identifier, following the normal rules for identifiers (unquoted case-insensitive, double-quoted case-sensitive). For backwards compatibility, some statements (e.g. isql CONNECT) accept usernames enclosed in single quotes, which will behave as normal, unquoted identifiers.

The maximum password length depends on the user manager plugin (parameter UserManager, in firebird.conf or databases.conf). Passwords are case-sensitive. The default user manager is the first plugin in the UserManager list, but this can be overridden in the SQL user management statements. For the Srp plugin, the maximum password length is 255 characters, for an effective length of 20 bytes (see also Why is the effective password length of SRP 20 bytes?). For the Legacy_UserManager plugin only the first eight bytes are significant; whilst it is valid to enter a password longer than eight bytes for Legacy_UserManager, any subsequent characters are ignored.

Why is the effective password length of SRP 20 bytes?

The SRP plugin does not actually have a 20 byte limit on password length, and longer passwords can be used. Hashes of different passwords longer than 20 bytes are also—usually—different. This effective limit comes from the limited hash length in SHA1 (used inside Firebirds SRP implementation), 20 bytes or 160 bits, and the “pigeonhole principle”. Sooner or later, there will be a shorter (or longer) password that has the same hash (e.g. in a brute force attack). That is why often the effective password length for the
The embedded version of the server does not use authentication. However, the username, and — if necessary — the role, must be specified in the connection parameters, as they control access to database objects.

SYSDBA or the owner of the database get unrestricted access to all objects of the database. Users with the RDB$ADMIN role get similar unrestricted access if they specify the role when connecting.

### 13.1.1. Specially Privileged Users

In Firebird, the SYSDBA account is a “Superuser” that exists beyond any security restrictions. It has complete access to all objects in all regular databases on the server, and full read/write access to the accounts in the security database `security3.fdb`. No user has access to the metadata of the security database.

For Srp, the SYSDBA account does not exist by default; it will need to be created using an embedded connection. For Legacy Auth, the default SYSDBA password on Windows and MacOS is “masterkey” — or “masterke”, to be exact, because of the 8-character length limit.

---

**Extremely Important!**

The default password “masterkey” is known across the universe. It should be changed as soon as the Firebird server installation is complete.

Other users can acquire elevated privileges in several ways, some of which are dependent on the operating system platform. These are discussed in the sections that follow and are summarised in [Administrators](#).

**POSIX Hosts**

On POSIX systems, including MacOS, the POSIX username will be used as the Firebird Embedded username if username is not explicitly specified.

**The SYSDBA User on POSIX**

On POSIX hosts, other than MacOSX, the SYSDBA user does not have a default password. If the full installation is done using the standard scripts, a one-off password will be created and stored in a text file in the same directory as `security3.fdb`, commonly `/opt/firebird/`. The name of the password file is `SYSDBA.password`.

In an installation performed by a distribution-specific installer, the location of the security database and the password file may be different from the standard one.

**The root User**

The `root` user can act directly as SYSDBA on Firebird Embedded. Firebird will treat `root` as though it were SYSDBA, and it provides access to all databases on the server.
Windows Hosts

On Windows server-capable operating systems, operating system accounts can be used. Windows authentication (also known as “trusted authentication”) can be enabled by including the Win_Sspi plugin in the AuthServer list in firebird.conf. The plugin must also be present in the AuthClient setting at the client-side.

Windows operating system administrators are not automatically granted SYSDBA privileges when connecting to a database. To make that happen, the internally-created role RDB$ADMIN must be altered by SYSDBA or the database owner, to enable it. For details, refer to the later section entitled AUTO ADMIN MAPPING.

Prior to Firebird 3.0, with trusted authentication enabled, users who passed the default checks were automatically mapped to CURRENT_USER. In Firebird 3.0 and later, the mapping must be done explicitly using CREATE MAPPING.

The Database Owner

The “owner” of a database is either the user who was CURRENT_USER at the time of creation (or restore) of the database or, if the USER parameter was supplied in the CREATE DATABASE statement, the specified user.

“Owner” is not a username. The user who is the owner of a database has full administrator privileges with respect to that database, including the right to drop it, to restore it from a backup and to enable or disable the AUTO ADMIN MAPPING capability.

Prior to Firebird 2.1, the owner had no automatic privileges over any database objects that were created by other users.

13.1.2. RDB$ADMIN Role

The internally-created role RDB$ADMIN is present in all databases. Assigning the RDB$ADMIN role to a regular user in a database grants that user the privileges of the SYSDBA, in that database only.

The elevated privileges take effect when the user is logged in to that regular database under the RDB$ADMIN role, and gives full control over all objects in that database.

Being granted the RDB$ADMIN role in the security database confers the authority to create, edit and delete user accounts.

In both cases, the user with the elevated privileges can assign RDB$ADMIN role to any other user. In other words, specifying WITH ADMIN OPTION is unnecessary because it is built into the role.

Granting the RDB$ADMIN Role in the Security Database

Since nobody — not even SYSDBA — can connect to the security database remotely, the GRANT and REVOKE statements are of no use for this task. Instead, the RDB$ADMIN role is granted and revoked using the SQL statements for user management:
CREATE USER new_user
    PASSWORD 'password'
    GRANT ADMIN ROLE;

ALTER USER existing_user
    GRANT ADMIN ROLE;

ALTER USER existing_user
    REVOKE ADMIN ROLE;

GRANT ADMIN ROLE and REVOKE ADMIN ROLE are not statements in the GRANT and REVOKE lexicon. They are three-word clauses to the statements CREATE USER and ALTER USER.

Table 219. Parameters for RDB$ADMIN Role GRANT and REVOKE

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_user</td>
<td>Name for the new user</td>
</tr>
<tr>
<td>existing_user</td>
<td>Name of an existing user</td>
</tr>
<tr>
<td>password</td>
<td>User password</td>
</tr>
</tbody>
</table>

The grantor must be logged in as an administrator.

See also
CREATE USER, ALTER USER, GRANT, REVOKE

Doing the Same Task Using gsec

With Firebird 3.0, gsec was deprecated. It is recommended to use the SQL user management statements instead.

An alternative is to use gsec with the -admin parameter to store the RDB$ADMIN attribute on the user’s record:

```
gsec -add new_user -pw password -admin yes
```
```
gsec -mo existing_user -admin yes
```
```
gsec -mo existing_user -admin no
```

Depending on the administrative status of the current user, more parameters may be needed when invoking gsec, e.g. -user and -pass, or -trusted.

Using the RDB$ADMIN Role in the Security Database

To manage user accounts through SQL, the grantee must specify the RDB$ADMIN role when connecting or through SET ROLE. No user can connect to the security database remotely, so the solution is that the user connects to a regular database where they also have RDB$ADMIN rights, supplying the RDB$ADMIN role in their login parameters. From there, they can submit any SQL user
management command.

If there is no regular database where the user has the RDB$ADMIN role, then account management via SQL queries is not possible, unless they connect directly to the security database using an embedded connection.

**Using `gsec` with RDB$ADMIN Rights**

To perform user management with `gsec`, the user must provide the extra switch `-role rdb$admin`.

**Granting the RDB$ADMIN Role in a Regular Database**

In a regular database, the RDB$ADMIN role is granted and revoked with the usual syntax for granting and revoking roles:

```sql
GRANT RDB$ADMIN TO username
REVOKE RDB$ADMIN FROM username
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>Name of the user</td>
</tr>
</tbody>
</table>

In order to grant and revoke the RDB$ADMIN role, the grantor must be logged in as an administrator.

See also

`GRANT`, `REVOKE`

**Using the RDB$ADMIN Role in a Regular Database**

To exercise their RDB$ADMIN privileges, the grantee has to include the role in the connection attributes when connecting to the database, or specify it later using `SET ROLE`.

**AUTO ADMIN MAPPING**

Windows Administrators are not automatically granted RDB$ADMIN privileges when connecting to a database (if Win_Sspi is enabled, of course) The AUTO ADMIN MAPPING switch now determines whether Administrators have automatic RDB$ADMIN rights, on a database-by-database basis. By default, when a database is created, it is disabled.

If AUTO ADMIN MAPPING is enabled in the database, it will take effect whenever a Windows Administrator connects:

a. using Win_Sspi authentication, and
b. without specifying any role

After a successful “auto admin” connection, the current role is set to RDB$ADMIN.

If an explicit role was specified on connect, the RDB$ADMIN role can be assumed later in the session.
Auto Admin Mapping in Regular Databases

To enable and disable automatic mapping in a regular database:

```sql
ALTER ROLE RDB$ADMIN
    SET AUTO ADMIN MAPPING;  -- enable it

ALTER ROLE RDB$ADMIN
    DROP AUTO ADMIN MAPPING; -- disable it
```

Either statement must be issued by a user with sufficient rights, that is:

- The database owner
- An administrator
- A user with the ALTER ANY ROLE privilege

The statement

```sql
ALTER ROLE RDB$ADMIN
    SET AUTO ADMIN MAPPING;
```

is a simplified form of a `CREATE MAPPING` statement to create a mapping of the predefined group `DOMAIN_ANY_RID_ADMINS` to the role of `RDB$ADMIN`:

```sql
CREATE MAPPING WIN_ADMINS
    USING PLUGIN WIN_SSPI
    FROM Predefined_Group DOMAIN_ANY_RID_ADMINS
    TO ROLE RDB$ADMIN;
```

Accordingly, the statement

```sql
ALTER ROLE RDB$ADMIN
    DROP AUTO ADMIN MAPPING
```

is equivalent to the statement

```sql
DROP MAPPING WIN_ADMINS;
```

For details, see Mapping of Users to Objects

In a regular database, the status of `AUTO ADMIN MAPPING` is checked only at connect time. If an Administrator has the `RDB$ADMIN` role because auto-mapping was on when they logged in, they will
keep that role for the duration of the session, even if they or someone else turns off the mapping in the meantime.

Likewise, switching on AUTO ADMIN MAPPING will not change the current role to RDB$ADMIN for Administrators who were already connected.

**Auto Admin Mapping in the Security Database**

The ALTER ROLE RDB$ADMIN statement cannot enable or disable AUTO ADMIN MAPPING in the security database. However, you can create a global mapping for the predefined group DOMAIN_ANY_RID_ADMINS to the role RDB$ADMIN in the following way:

```
CREATE GLOBAL MAPPING WIN_ADMINS
  USING PLUGIN WIN_SSPI
  FROM Predefined_Group DOMAIN_ANY_RID_ADMINS
  TO ROLE RDB$ADMIN;
```

Additionally, you can use `gsec`:

```
gsec -mapping set

 gsec -mapping drop
```

Depending on the administrative status of the current user, more parameters may be needed when invoking `gsec`, e.g. `-user` and `-pass`, or `-trusted`.

Only SYSDBA can enable AUTO ADMIN MAPPING if it is disabled, but any administrator can turn it off.

When turning off AUTO ADMIN MAPPING in `gsec`, the user turns off the mechanism itself which gave them access, and thus they would not be able to re-enable AUTO ADMIN MAPPING. Even in an interactive `gsec` session, the new flag setting takes effect immediately.

### 13.1.3. Administrators

As a general description, an administrator is a user that has sufficient rights to read, write to, create, alter or delete any object in a database to which that user’s administrator status applies. The table summarises how “Superuser” privileges are enabled in the various Firebird security contexts.

<table>
<thead>
<tr>
<th>User</th>
<th>RDB$ADMIN Role</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSDBA</td>
<td>Auto</td>
<td>Exists automatically at server level. Has full privileges to all objects in all databases. Can create, alter and drop users, but has no direct remote access to the security database</td>
</tr>
<tr>
<td>root user on POSIX</td>
<td>Auto</td>
<td>Exactly like SYSDBA. Firebird Embedded only.</td>
</tr>
</tbody>
</table>
### User Access Levels

<table>
<thead>
<tr>
<th>User</th>
<th>RDB$ADMIN Role</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superuser on POSIX</td>
<td>Auto</td>
<td>Exactly like SYSDBA. Firebird Embedded only.</td>
</tr>
</tbody>
</table>
| Windows Administrator | Set as CURRENT_ROLE if login succeeds | Exactly like SYSDBA if all of the following are true:  
  - In firebird.conf file, AuthServer includes Win_Sspi, and Win_Sspi is present in the client-side plugins (AuthClient) configuration  
  - In databases where AUTO_ADMIN MAPPING is enabled, or an equivalent mapping of the predefined group DOMAIN_ANY_RID_ADMINs for the role RDB$ADMIN exists  
  - No role is specified at login |
| Database owner   | Auto           | Like SYSDBA, but only in the databases they own                        |
| Regular user     | Must be previously granted; must be supplied at login | Like SYSDBA, but only in the databases where the role is granted     |
| POSIX OS user    | Must be previously granted; must be supplied at login | Like SYSDBA, but only in the databases where the role is granted. Firebird Embedded only. |
| Windows user     | Must be previously granted; must be supplied at login | Like SYSDBA, but only in the databases where the role is granted. Only available if in firebird.conf file, AuthServer includes Win_Sspi, and Win_Sspi is present in the client-side plugins (AuthClient) configuration |

### 13.2. SQL Statements for User Management

This section describes the SQL statements for creating, modifying and deleting Firebird user accounts. These statements can be executed by the following users:

- **SYSDBA**

- Any user with the RDB$ADMIN role in the security database, and the RDB$ADMIN role in the database of the active connection (the user must specify the RDB$ADMIN role on connect, or using SET ROLE)

- When the AUTO_ADMIN_MAPPING flag is enabled in the security database (security3.fdb or whatever is the security database configured for the current database in the databases.conf), any Windows Administrator - assuming Win_Sspi was used to connect without specifying roles.

For a Windows Administrator, AUTO_ADMIN_MAPPING enabled only in a regular database is not sufficient to permit management of other users. For instructions to enable it in the security database, see **Auto Admin Mapping in the Security Database**.
Non-privileged users can use only the \texttt{ALTER USER} statement, and then only to edit some data of their own account.

\subsection*{13.2.1. \texttt{CREATE USER}}

\textit{Used for}

Creating a Firebird user account

\textit{Available in}

DSQL

\textit{Syntax}

```
CREATE USER \texttt{username}
\texttt{<user_option> \texttt{[\texttt{<user_option> \ldots}]} [\texttt{T\texttt{AGS} \texttt{\texttt{\} <user_var> [, <user_var> \ldots]}]}]
```

\texttt{<user_option> ::=}

\begin{itemize}
\item \texttt{PASSWORD \texttt{'}password\texttt{'}}
\item \texttt{FIRSTNAME \texttt{'}firstname\texttt{'}}
\item \texttt{MIDDLENAME \texttt{'}middlename\texttt{'}}
\item \texttt{LASTNAME \texttt{'}lastname\texttt{'}}
\item \{\texttt{GRANT | REVOKE}\} \texttt{ADMIN ROLE}
\item \{\texttt{ACTIVE | INACTIVE}\}
\item \texttt{USING PLUGIN \texttt{plugin_name}}
\end{itemize}

\texttt{<user_var> ::=}

\begin{itemize}
\item \texttt{tag_name = 'tag_value'}
\item \texttt{DROP tag_name}
\end{itemize}

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Parameter} & \textbf{Description} \\
\hline
\texttt{username} & Username. The maximum length is 31 characters, following the rules for Firebird identifiers. \\
\hline
\texttt{password} & User password. Valid or effective password length depends on the user manager plugin. Case-sensitive. \\
\hline
\texttt{firstname} & Optional: User's first name. Maximum length 31 characters \\
\hline
\texttt{middlename} & Optional: User's middle name. Maximum length 31 characters \\
\hline
\texttt{lastname} & Optional: User's last name. Maximum length 31 characters. \\
\hline
\texttt{plugin_name} & Name of the user manager plugin. \\
\hline
\texttt{tag_name} & Name of a custom attribute. The maximum length is 31 characters, following the rules for Firebird regular identifiers. \\
\hline
\texttt{tag_value} & Value of the custom attribute. The maximum length is 255 characters. \\
\hline
\end{tabular}
\end{table}

The \texttt{CREATE USER} statement creates a new Firebird user account. If the user already exist in the
Firebird security database for the specified user manager plugin, an appropriate error be raised. It is possible to create multiple users with the same name: one per user manager plugin.

The *username* argument must follow the rules for Firebird regular identifiers: see *Identifiers* in the *Structure* chapter. Usernames are case-sensitive when double-quoted (in other words, they follow the same rules as other delimited identifiers).

Since Firebird 3.0, usernames follow the general naming conventions of identifiers. Thus, a user named "Alex" is distinct from a user named "ALEX"

```
CREATE USER ALEX PASSWORD 'bz23ds';
- this user is the same as the first one
CREATE USER Alex PASSWORD 'bz23ds';
- this user is the same as the first one
CREATE USER "ALEX" PASSWORD 'bz23ds';
- and this is a different user
CREATE USER "Alex" PASSWORD 'bz23ds';
```

The *PASSWORD* clause specifies the user's password, and is required. The valid or effective password length depends on the user manager plugin, see also *User Authentication*.

The optional *FIRSTNAME*, *MIDDLENAME* and *LASTNAME* clauses can be used to specify additional user properties, such as the person's first name, middle name and last name, respectively. They are just simple `VARCHAR(31)` fields and can be used to store anything you prefer.

If the *GRANT ADMIN ROLE* clause is specified, the new user account is created with the privileges of the `RDB$ADMIN` role in the security database (`security3.fdb` or database-specific). It allows the new user to manage user accounts from any regular database they log into, but it does not grant the user any special privileges on objects in those databases.

The *REVOKE ADMIN ROLE* clause is syntactically valid in a *CREATE USER* statement, but has no effect. It is not possible to specify *GRANT ADMIN ROLE* and *REVOKE ADMIN ROLE* in one statement.

The *ACTIVE* clause specifies the user is active and can log in, this is the default.

The *INACTIVE* clause specifies the user is inactive and cannot log in. It is not possible to specify *ACTIVE* and *INACTIVE* in one statement. The *ACTIVE/INACTIVE* option is not supported by the *Legacy_UsersManager* and will be ignored.

The *USING PLUGIN* clause explicitly specifies the user manager plugin to use for creating the user. Only plugins listed in the *UserManager* configuration for this database (firebird.conf, or overridden in databases.conf) are valid. The default user manager (first in the *UserManager* configuration) is applied when this clause is not specified.

Users of the same name created using different user manager plugins are different objects. Therefore, the user created with one user manager plugin can only be
altered or dropped by that same plugin.

From the perspective of ownership, and privileges and roles granted in a databases, different user objects with the same name are considered one and the same user.

The TAGS clause can be used to specify additional user attributes. Custom attributes are not supported (silently ignored) by the Legacy_UserManager. Custom attributes names follow the rules of Firebird identifiers, but are handled case-insensitive (for example, specifying both "A BC" and "a bc" will raise an error). The value of a custom attribute can be a string of maximum 255 characters. The DROP tag_name option is syntactically valid in CREATE USER, but behaves as if the property is not specified.

![Warning] Users can view and alter their own custom attributes.

CREATE/ALTER/DROP USER are DDL statements, and only take effect at commit. Remember to COMMIT your work. In isql, the command SET AUTO ON will enable autocommit on DDL statements. In third-party tools and other user applications, this may not be the case.

Who Can Create a User

To create a user account, the current user must have administrator privileges in the security database. Administrator privileges only in regular databases are not sufficient.

CREATE USER Examples

1. Creating a user with the username bigshot:

   CREATE USER bigshot PASSWORD 'buckshot';

2. Creating a user with the Legacy_UserManager user manager plugin

   CREATE USER godzilla PASSWORD 'robot'
   USING PLUGIN Legacy_UserManager;

3. Creating the user john with custom attributes:

   CREATE USER john PASSWORD 'fYe_3Ksw'
   FIRSTNAME 'John' LASTNAME 'Doe'
   TAGS (BIRTHYEAR='1970', CITY='New York');

4. Creating an inactive user:

   CREATE USER john PASSWORD 'fYe_3Ksw'
5. Creating the user `superuser` with user management privileges:

```sql
CREATE USER superuser PASSWORD 'kMn8Kjh'
GRANT ADMIN ROLE;
```

See also

`ALTER USER, CREATE OR ALTER USER, DROP USER`

### 13.2.2. ALTER USER

*Used for*

Modifying a Firebird user account

*Available in*

DSQL

*Syntax*

```
ALTER {USER username | CURRENT USER}
[SET] [<user_option> [<user_option> ...]]
[TAGS (<user_var> [, <user_var> ...])]
```

```
<user_option> ::= 
  PASSWORD 'password'
  | FIRSTNAME 'firstname'
  | MIDDLENAME 'middlename'
  | LASTNAME 'lastname'
  | {GRANT | REVOKE} ADMIN ROLE
  | {ACTIVE | INACTIVE}
  | USING PLUGIN plugin_name
```

```
<user_var> ::= 
  tag_name = 'tag_value'
  | DROP tag_name
```

See `CREATE USER` for details on the statement parameters.

The `ALTER USER` statement changes the details in the named Firebird user account. The `ALTER USER` statement must contain at least one of the optional clauses other than `USING PLUGIN`.

Any user can alter his or her own account, except that only an administrator may use `GRANT/REVOKE ADMIN ROLE` and `ACTIVE/INACTIVE`.

All clauses are optional, but at least one other than `USING PLUGIN` must be present:

- The `PASSWORD` parameter is for changing the password for the user
FIRSTNAME, MIDDLENAME and LASTNAME update these optional user properties, such as the person’s first name, middle name and last name respectively.

GRANT ADMIN ROLE grants the user the privileges of the RDB$ADMIN role in the security database (security3.fdb), enabling them to manage the accounts of other users. It does not grant the user any special privileges in regular databases.

REVOKE ADMIN ROLE removes the user’s administrator in the security database which, once the transaction is committed, will deny that user the ability to alter any user account except their own.

ACTIVE will enable a disabled account (not supported for Legacy_UserManager).

INACTIVE will disable an account (not supported for Legacy_UserManager). This is convenient to temporarily disable an account without deleting it.

USING PLUGIN specifies the user manager plugin to use.

TAGS can be used to add, update or remove (DROP) additional custom attributes (not supported for Legacy_UserManager). Attributes not listed will not be changed.

See CREATE_USER for more details on the clauses.

If you need to change your own account, then instead of specifying the name of the current user, you can use the CURRENT_USER clause.

The ALTER_CURRENT_USER statement follows the normal rules for selecting the user manager plugin. If the current user was created with a non-default user manager plugin, they will need to explicitly specify the user manager plugins with USING_PLUGIN plugin_name, or they will receive an error that the user is not found. Or, if a user with the same name exists for the default user manager, they will alter that user instead.

Remember to commit your work if you are working in an application that does not auto-commit DDL.

Who Can Alter a User?

To modify the account of another user, the current user must have administrator privileges in the security database. Anyone can modify their own account, except for the GRANT/REVOKE ADMIN ROLE and ACTIVE/INACTIVE options, which require administrative privileges to change.

ALTER_USER Examples

1. Changing the password for the user bobby and granting them user management privileges:

   ALTER USER bobby PASSWORD '67-UiT_G8'
   GRANT ADMIN ROLE;

2. Editing the optional properties (the first and last names) of the user dan:
ALTER USER dan
FIRSTNAME 'No_Jack'
LASTNAME 'Kennedy';

3. Revoking user management privileges from user dumbbell:

ALTER USER dumbbell
DROP ADMIN ROLE;

See also
CREATE USER, DROP USER

13.2.3. CREATE OR ALTER USER

Used for
Creating a new or modifying an existing Firebird user account

Available in
DSQL

Syntax

CREATE OR ALTER USER username
   [SET] [<user_option> [<user_option> ...]]
   [TAGS (<user_var> [, <user_var> ...])]

<user_option> ::=  
   PASSWORD 'password'
   | FIRSTNAME 'firstname'
   | MIDDLENAME 'middlename'
   | LASTNAME 'lastname'
   | {GRANT | REVOKE} ADMIN ROLE
   | {ACTIVE | INACTIVE}
   | USING PLUGIN plugin_name

<user_var> ::= 
   tag_name = 'tag_value'
   | DROP tag_name

See CREATE USER and ALTER USER for details on the statement parameters.

The CREATE OR ALTER USER statement creates a new or changes the details in the named Firebird user account. If the user does not exist, it will be created as if executing the CREATE USER statement. If the user already exists, it will be modified as if executing the ALTER USER statement. The CREATE OR ALTER USER statement must contain at least one of the optional clauses other than USING PLUGIN. If the user does not exist yet, the PASSWORD clause is required.
Remember to commit your work if you are working in an application that does not auto-commit DDL.

**CREATE OR ALTER USER Examples**

Creating or altering a user

```
CREATE OR ALTER USER john PASSWORD 'fYe_3Ksw'
FIRSTNAME 'John'
LASTNAME 'Doe'
INACTIVE;
```

See also

**CREATE USER, ALTER USER, DROP USER**

### 13.2.4. DROP USER

**Used for**
Deleting a Firebird user account

**Available in**
DSQL

**Syntax**

```
DROP USER username
[USING PLUGIN plugin_name]
```

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>Username</td>
</tr>
<tr>
<td>plugin_name</td>
<td>Name of the user manager plugin</td>
</tr>
</tbody>
</table>

The **DROP USER** statement deletes a Firebird user account.

The optional **USING PLUGIN** clause explicitly specifies the user manager plugin to use for dropping the user. Only plugins listed in the **UserManager** configuration for this database (firebird.conf, or overridden in databases.conf) are valid. The default user manager (first in the **UserManager** configuration) is applied when this clause is not specified.

Users of the same name created using different user manager plugins are different objects. Therefore, the user created with one user manager plugin can only be dropped by that same plugin.

Remember to commit your work if you are working in an application that does not auto-commit DDL.
Who Can Drop a User?

To drop a user, the current user must have administrator privileges.

DROP USER Example

1. Deleting the user bobby:

   ```sql
   DROP USER bobby;
   ```

2. Removing a user created with the Legacy_UserManager plugin:

   ```sql
   DROP USER Godzilla
   USING PLUGIN Legacy_UserManager;
   ```

See also

CREATE USER, ALTER USER

13.3. SQL Privileges

The second level of Firebird’s security model is SQL privileges. Whilst a successful login — the first level — authorises a user’s access to the server and to all databases under that server, it does not imply that the user has access to any objects in any databases. When an object is created, only the user that created it (its owner) and administrators have access to it. The user needs privileges on each object they need to access. As a general rule, privileges must be granted explicitly to a user by the object owner or an administrator of the database.

A privilege comprises a DML access type (SELECT, INSERT, UPDATE, DELETE, EXECUTE and REFERENCES), the name of a database object (table, view, procedure, role) and the name of the grantee (user, procedure, trigger, role). Various means are available to grant multiple types of access on an object to multiple users in a single GRANT statement. Privileges may be revoked from a user with REVOKE statements.

An additional type of privileges, DDL privileges, provide rights to create, alter or drop specific types of metadata objects.

Privileges are stored in the database to which they apply and are not applicable to any other database, except the DATABASE DDL privileges, which are stored in the security database.

13.3.1. The Object Owner

The user who created a database object becomes its owner. Only the owner of an object and users with administrator privileges in the database, including the database owner, can alter or drop the database object.

Administrators, the database owner or the object owner can grant privileges to and revoke them from other users, including privileges to grant privileges to other users. The process of granting and
revoking SQL privileges is implemented with two statements, **GRANT** and **REVOKE**.

### 13.4. ROLE

A *role* is a database object that packages a set of *privileges*. Roles implement the concept of access control at a group level. Multiple privileges are granted to the role and then that role can be granted to or revoked from one or many users.

A user that is granted a role must supply that role in their login credentials in order to exercise the associated privileges. Any other privileges granted to the user directly are not affected by their login with the role. Logging in with multiple roles simultaneously is not supported.

In this section the tasks of creating and dropping roles are discussed.

#### 13.4.1. CREATE ROLE

*Used for*
Creating a new ROLE object

*Available in*
DSQL, ESQL

*Syntax*

```sql
CREATE ROLE rolename
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rolename</td>
<td>Role name. The maximum length is 31 characters</td>
</tr>
</tbody>
</table>

The statement **CREATE ROLE** creates a new role object, to which one or more privileges can be granted subsequently. The name of a role must be unique among the names of roles in the current database.

It is advisable to make the name of a role unique among usernames as well. The system will not prevent the creation of a role whose name clashes with an existing username but, if it happens, the user will be unable to connect to the database.

*Who Can Create a Role*

The **CREATE ROLE** statement can be executed by:

- **Administrators**
- Users with the **CREATE ROLE** privilege

The user executing the **CREATE ROLE** statement becomes the owner of the role.
CREATE ROLE Example

Creating a role named SELLERS

CREATE ROLE SELLERS;

See also
DROP ROLE, GRANT, REVOKE

13.4.2. ALTER ROLE

Used for
Altering a role (enabling or disabling auto-admin mapping)

Available in
DSQL

Syntax

```
ALTER ROLE rolename
{SET | DROP} AUTO ADMIN MAPPING
```

Table 225. ALTER ROLE Statement Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rolename</td>
<td>Role name; specifying anything other than RDB$ADMIN will fail</td>
</tr>
</tbody>
</table>

ALTER ROLE has no place in the create-alter-drop paradigm for database objects since a role has no attributes that can be modified. Its actual effect is to alter an attribute of the database: Firebird uses it to enable and disable the capability for Windows Administrators to assume administrator privileges automatically when logging in.

This capability can affect only one role: the system-generated role RDB$ADMIN that exists in every database of ODS 11.2 or higher. Several factors are involved in enabling this feature.

For details, see AUTO ADMIN MAPPING.

Who Can Alter a Role

The ALTER ROLE statement can be executed by:

- Administrators

Although an ALTER ANY ROLE DDL privilege exists, it does not apply because creating or dropping mappings requires administrator privileges.
13.4.3. DROP ROLE

*Used for*
Deleting a role

*Available in*
DSQL, ESQL

**Syntax**

```
DROP ROLE rolename
```

The statement DROP ROLE deletes an existing role. It takes just a single argument, the name of the role. Once the role is deleted, the entire set of privileges is revoked from all users and objects that were granted the role.

**Who Can Drop a Role**

The DROP ROLE statement can be executed by:

- Administrators
- The owner of the role
- Users with the DROP ANY ROLE privilege

**DROP ROLE Examples**

*Deleting the role SELLERS*

```
DROP ROLE SELLERS;
```

*See also*

CREATE ROLE, GRANT, REVOKE

13.5. Statements for Granting Privileges

A GRANT statement is used for granting privileges—including roles—to users and other database objects.

13.5.1. GRANT

*Used for*
Granting privileges and assigning roles

*Available in*
DSQL, ESQL
Syntax (granting privileges)

GRANT <privileges>
    TO <grantee_list>
    [WITH GRANT OPTION]
    [{GRANTED BY | AS} [USER] grantor]

<privileges> ::= <table_privileges> | <execute_privilege>
    | <usage_privilege> | <ddl_privileges>
    | <db_ddl_privilege>

<table_privileges> ::= {ALL [PRIVILEGES] | <table_privilege_list> }
    ON [TABLE] {table_name | view_name}

<table_privilege_list> ::= <table_privilege> [, <table_privilege> ...]

<table_privilege> ::= SELECT | DELETE | INSERT
    | UPDATE [(col[, col ...])]]
    | REFERENCES [(col[, col ...])]

<execute_privilege> ::= EXECUTE ON
    { PROCEDURE proc_name | FUNCTION func_name
    | PACKAGE package_name }

<usage_privilege> ::= USAGE ON
    { EXCEPTION exception_name
    | {GENERATOR | SEQUENCE} sequence_name }

<ddl_privileges> ::= {ALL [PRIVILEGES] | <ddl_privilege_list>} <object_type>

<ddl_privilege_list> ::= <ddl_privilege> [, <ddl_privilege> ...]

<ddl_privilege> ::= CREATE | ALTER ANY | DROP ANY

<object_type> ::= CHARACTER SET | COLLATION | DOMAIN | EXCEPTION
    | FILTER | FUNCTION | GENERATOR | PACKAGE
    | PROCEDURE | ROLE | SEQUENCE | TABLE | VIEW

<db_ddl_privileges> ::= {ALL [PRIVILEGES] | <db_ddl_privilege_list>} {DATABASE | SCHEMA}

<db_ddl_privilege_list> ::= <db_ddl_privilege> [, <db_ddl_privilege> ...]
**Syntax (granting roles)**

```
GRANT <role_granted>
  TO <role_grantee_list>
  [WITH ADMIN OPTION]
  [[{GRANTED BY | AS} [USER] grantor]
```

```
<role_granted> ::= role_name [, role_name ...]
```

```
<role_grantee_list> ::=<role_grantee> [, <role_grantee> ...]
```

```
<role_grantee> ::= [USER] username
```

**Table 226. GRANT Statement Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantor</td>
<td>The user granting the privilege(s)</td>
</tr>
<tr>
<td>table_name</td>
<td>The name of a table</td>
</tr>
<tr>
<td>view_name</td>
<td>The name of a view</td>
</tr>
<tr>
<td>col</td>
<td>The name of table column</td>
</tr>
<tr>
<td>proc_name</td>
<td>The name of a stored procedure</td>
</tr>
<tr>
<td>func_name</td>
<td>The name of a stored function (or UDF)</td>
</tr>
<tr>
<td>package_name</td>
<td>The name of a package</td>
</tr>
<tr>
<td>exception_name</td>
<td>The name of an exception</td>
</tr>
<tr>
<td>sequence_name</td>
<td>The name of a sequence (generator)</td>
</tr>
<tr>
<td>object_type</td>
<td>The type of metadata object</td>
</tr>
<tr>
<td>trig_name</td>
<td>The name of a trigger</td>
</tr>
<tr>
<td>role_name</td>
<td>Role name</td>
</tr>
<tr>
<td>username</td>
<td>The username to which the privileges are granted to or to which the role</td>
</tr>
<tr>
<td></td>
<td>is assigned. If the USER keyword is absent, it can also be a role.</td>
</tr>
<tr>
<td>Unix_group</td>
<td>The name of a user group in a POSIX operating system</td>
</tr>
</tbody>
</table>
The GRANT statement grants one or more privileges on database objects to users, roles, or other database objects.

A regular, authenticated user has no privileges on any database object until they are explicitly granted, either to that individual user or to all users bundled as the user PUBLIC. When an object is created, only its creator (the owner) and administrators have privileges to it, and can grant privileges to other users, roles, or objects.

Different sets of privileges apply to different types of metadata objects. The different types of privileges will be described separately later in this section.

The TO Clause

The TO clause specifies the users, roles, and other database objects that are to be granted the privileges enumerated in privileges. The clause is mandatory.

The optional USER keyword in the TO clause allow you to specify exactly who or what is granted the privilege. If a USER (or ROLE) keyword is not specified, the server first checks for a role with this name and, if there is no such role, the privileges are granted to the user with that name without further checking.

It is recommended to always explicitly specify USER and ROLE to avoid ambiguity. Future versions of Firebird may make USER mandatory.

- When a GRANT statement is executed, the security database is not checked for the existence of the grantee user. This is not a bug: SQL permissions are concerned with controlling data access for authenticated users, both native and trusted, and trusted operating system users are not stored in the security database.
- When granting a privilege to a database object other than user or role, such as a procedure, trigger or view, you must specify the object type.
- Although the USER keyword is optional, it is advisable to use it, in order to avoid ambiguity with roles.

Packaging Privileges in a ROLE Object

A role is a “container” object that can be used to package a collection of privileges. Use of the role is then granted to each user that requires those privileges. A role can also be granted to a list of users.

The role must exist before privileges can be granted to it. See CREATE ROLE for the syntax and rules. The role is maintained by granting privileges to it and, when required, revoking privileges from it. When a role is dropped (see DROP ROLE), all users lose the privileges acquired through the role. Any privileges that were granted additionally to an affected user by way of a different grant statement are retained.
A user that is granted a role must supply that role with his login credentials in order to exercise the associated privileges. Any other privileges granted to the user are not affected by logging in with a role.

More than one role can be granted to the same user but logging in with multiple roles simultaneously is not supported.

A role can be granted only to a user.

**The User PUBLIC**

Firebird has a predefined user named PUBLIC, that represents all users. Privileges for operations on a particular object that are granted to the user PUBLIC can be exercised by any authenticated user.

> If privileges are granted to the user PUBLIC, they should be revoked from the user PUBLIC as well.

**The WITH GRANT OPTION Clause**

The optional WITH GRANT OPTION clause allows the users specified in the user list to grant the privileges specified in the privilege list to other users.

> It is possible to assign this option to the user PUBLIC. Do not do this!

**The GRANTED BY Clause**

By default, when privileges are granted in a database, the current user is recorded as the grantor. The GRANTED BY clause enables the current user to grant those privileges as another user.

When using the REVOKE statement, it will fail if the current user is not the user that was named in the GRANTED BY clause.

The GRANTED BY (and AS) clause can be used only by the database owner and other administrators. The object owner cannot use GRANTED BY unless they also have administrator privileges.

**Alternative Syntax Using AS username**

The non-standard AS clause is supported as a synonym of the GRANTED BY clause to simplify migration from other database systems.

**Privileges on Tables and Views**

For tables and views, unlike other metadata objects, it is possible to grant several privileges at once.

**List of Privileges on Tables**

**SELECT**

Permits the user or object to SELECT data from the table or view

**INSERT**

Permits the user or object to INSERT rows into the table or view
DELETE
Permits the user or object to DELETE rows from the table or view

UPDATE
Permits the user or object to UPDATE rows in the table or view, optionally restricted to specific columns

REFERENCES
Permits the user or object to reference the table via a foreign key, optionally restricted to the specified columns. If the primary or unique key referenced by the foreign key of the other table is composite then all columns of the key must be specified.

ALL [PRIVILEGES]
Combines SELECT, INSERT, UPDATE, DELETE and REFERENCES privileges in a single package

Examples of GRANT <privilege> on Tables

1. SELECT and INSERT privileges to the user ALEX:

   GRANT SELECT, INSERT ON TABLE SALES TO USER ALEX;

2. The SELECT privilege to the MANAGER, ENGINEER roles and to the user IVAN:

   GRANT SELECT ON TABLE CUSTOMER
   TO ROLE MANAGER, ROLE ENGINEER, USER IVAN;

3. All privileges to the ADMINISTRATOR role, together with the authority to grant the same privileges to others:

   GRANT ALL ON TABLE CUSTOMER
   TO ROLE ADMINISTRATOR
   WITH GRANT OPTION;

4. The SELECT and REFERENCES privileges on the NAME column to all users and objects:

   GRANT SELECT, REFERENCES (NAME) ON TABLE COUNTRY
   TO PUBLIC;

5. The SELECT privilege being granted to the user IVAN by the user ALEX:

   GRANT SELECT ON TABLE EMPLOYEE
   TO USER IVAN
   GRANTED BY ALEX;
6. Granting the UPDATE privilege on the FIRST_NAME, LAST_NAME columns:

```
GRANT UPDATE (FIRST_NAME, LAST_NAME) ON TABLE EMPLOYEE
    TO USER IVAN;
```

7. Granting the INSERT privilege to the stored procedure ADD_EMP_PROJ:

```
GRANT INSERT ON EMPLOYEE_PROJECT
    TO PROCEDURE ADD_EMP_PROJ;
```

**The EXECUTE Privilege**

The EXECUTE privilege applies to stored procedures, stored functions (including UDFs), and packages. It allows the grantee to execute the specified object, and, if applicable, to retrieve its output.

In the case of selectable stored procedures, it acts somewhat like a SELECT privilege, insofar as this style of stored procedure is executed in response to a SELECT statement.

For packages, the EXECUTE privilege can only be granted for the package as a whole, not for individual subroutines.

**Examples of Granting the EXECUTE Privilege**

1. Granting the EXECUTE privilege on a stored procedure to a role:

```
GRANT EXECUTE ON PROCEDURE ADD_EMP_PROJ
    TO ROLE MANAGER;
```

2. Granting the EXECUTE privilege on a stored function to a role:

```
GRANT EXECUTE ON FUNCTION GET_BEGIN_DATE
    TO ROLE MANAGER;
```

3. Granting the EXECUTE privilege on a package to user PUBLIC:

```
GRANT EXECUTE ON PACKAGE APP_VAR
    TO USER PUBLIC;
```

4. Granting the EXECUTE privilege on a function to a package:

```
GRANT EXECUTE ON FUNCTION GET_BEGIN_DATE
    TO PACKAGE APP_VAR;
```
The USAGE Privilege

To be able to use metadata objects other than tables, views, stored procedures or functions, triggers and packages, it is necessary to grant the user (or database object like trigger, procedure or function) the USAGE privilege on these objects.

Since Firebird executes stored procedures and functions, triggers, and package routines with the privileges of the caller, it is necessary that either the user or otherwise the routine itself has been granted the USAGE privilege.

In Firebird 3.0, the USAGE privilege is only available for exceptions and sequences (in gen_id(gen_name, n) or ‘next value for gen_name). Support for the USAGE privilege for other metadata objects may be added in future releases.

For sequences (generators), the USAGE privilege only grants the right to increment the sequence using the GEN_ID function or NEXT VALUE FOR. The SET GENERATOR statement is a synonym for ALTER SEQUENCE … RESTART WITH …, and is considered a DDL statement. By default, only the owner of the sequence and administrators have the rights to such operations. The right to set the initial value of any sequence can be granted with GRANT ALTER ANY SEQUENCE, which is not recommend for general users.

Examples of Granting the USAGE Privilege

1. Granting the USAGE privilege on a sequence to a role:

   ```
   GRANT USAGE ON SEQUENCE GEN_AGE
   TO ROLE MANAGER;
   ```

2. Granting the USAGE privilege on a sequence to a trigger:

   ```
   GRANT USAGE ON SEQUENCE GEN_AGE
   TO TRIGGER TR_AGE_BI;
   ```

3. Granting the USAGE privilege on an exception to a package:

   ```
   GRANT USAGE ON EXCEPTION
   TO PACKAGE PKG_BILL;
   ```

DDL Privileges

By default, only administrators can create new metadata objects; altering or dropping these objects is restricted to the owner of the object (its creator) and administrators. DDL privileges can be used to grant privileges for these operations to other users.

Available DDL Privileges
CREATE
   Allows creation of an object of the specified type

ALTER ANY
   Allows modification of any object of the specified type

DROP ANY
   Allows deletion of any object of the specified type

ALL [PRIVILEGES]
   Combines the CREATE, ALTER ANY and DROP ANY privileges for the specified type

There are no separate DDL privileges for triggers and indexes. The necessary privileges are inherited from the table or view. Creating, altering or dropping a trigger or index requires the ALTER ANY TABLE or ALTER ANY VIEW privilege.

Examples of Granting DDL Privileges

1. Allow user JOE to create tables

   GRANT CREATE TABLE
       TO USER Joe;

2. Allow user JOE to alter any procedure

   GRANT ALTER ANY PROCEDURE
       TO USER Joe;

Database DDL Privileges

The syntax for granting privileges to create, alter or drop a database deviates from the normal syntax of granting DDL privileges for other object types.

Available Database DDL Privileges

CREATE
   Allows creation of a database

ALTER
   Allows modification of the current database

DROP
   Allows deletion of the current database

ALL [PRIVILEGES]
   Combines the ALTER and DROP privileges. ALL does not include the CREATE privilege.
The `ALTER DATABASE` and `DROP DATABASE` privileges apply only to the current database, whereas DDL privileges `ALTER ANY` and `DROP ANY` on other object types apply to all objects of the specified type in the current database. The privilege to alter or drop the current database can only be granted by administrators.

The `CREATE DATABASE` privilege is a special kind of privilege as it is saved in the security database. A list of users with the `CREATE DATABASE` privilege is available from the virtual table `SEC$DB_CREATORS`. Only administrators in the security database can grant the privilege to create a new database.

```
-- SCHEMA is currently a synonym for DATABASE; this may change in a future version, so we recommend to always use DATABASE
```

### Examples of Granting Database DDL Privileges

1. Granting `SUPERUSER` the privilege to create databases:
   ```sql
   GRANT CREATE DATABASE
   TO USER Superuser;
   ```

2. Granting `JOE` the privilege to execute `ALTER DATABASE` for the current database:
   ```sql
   GRANT ALTER DATABASE
   TO USER Joe;
   ```

3. Granting `FEDOR` the privilege to drop the current database:
   ```sql
   GRANT DROP DATABASE
   TO USER Fedor;
   ```

### Assigning Roles

Assigning a role is similar to granting a privilege. One or more roles can be assigned to one or more users, including the `user PUBLIC`, using one `GRANT` statement.

#### The `WITH ADMIN OPTION` Clause

The optional `WITH ADMIN OPTION` clause allows the users specified in the user list to grant the role(s) specified to other users.

```
It is possible to assign this option to PUBLIC. Do not do this!
```

### Examples of Role Assignment

1. Assigning the `DIRECTOR` and `MANAGER` roles to the user `IVAN`:
   ```sql
   GRANT DIRECTOR, MANAGER
   ```
2. Assigning the MANAGER role to the user ALEX with the authority to assign this role to other users:

    GRANT MANAGER TO USER ALEX WITH ADMIN OPTION;

See also

REVOKE

13.6. Statements for Revoking Privileges

A REVOKE statement is used for revoking privileges—including roles—from users and other database objects.

13.6.1. REVOKE

*Used for*
Revoking privileges or role assignments

*Available in*
DSQL, ESQL

*Syntax (revoking privileges)*

```
REVOKE [GRANT OPTION FOR] <privileges>
    FROM <grantee_list>
    [{GRANTED BY | AS} [USER] grantor]

<privileges> ::= ! See GRANT syntax !!
```

*Syntax (revoking roles)*

```
REVOKE [ADMIN OPTION FOR] <role_granted>
    FROM <role_grantee_list>
    [{GRANTED BY | AS} [USER] grantor]

<role_granted> ::= ! See GRANT syntax !!

<role_grantee_list> ::= ! See GRANT syntax !!
```

*Syntax (revoking all)*

```
REVOKE ALL ON ALL FROM <grantee_list>
```
The REVOKE statement revokes privileges that were granted using the GRANT statement from users, roles, and other database objects. See GRANT for detailed descriptions of the various types of privileges.

Only the user who granted the privilege can revoke it.

**The FROM Clause**

The FROM clause specifies a list of users, roles and other database objects that will have the enumerated privileges revoked. The optional USER keyword in the FROM clause allow you to specify exactly which type is to have the privilege revoked. If a USER (or ROLE) keyword is not specified, the server first checks for a role with this name and, if there is no such role, the privileges are revoked from the user with that name without further checking.

- Although the USER keyword is optional, it is advisable to use them in order to avoid ambiguity with roles.
- The REVOKE statement does not check for the existence of the user from which the privileges are being revoked.
- When revoking a privilege from a database object other than USER or ROLE, you must specify its object type.

**Revoking Privileges from user PUBLIC**

Privileges that were granted to the special user named PUBLIC must be revoked from the user PUBLIC. User PUBLIC provides a way to grant privileges to all users at once, but it is not “a group of users”.

**Revoking the GRANT OPTION**

The optional GRANT OPTION FOR clause revokes the user’s privilege to grant the specified privileges to other users, roles, or database objects (as previously granted with the WITH GRANT OPTION). It does not revoke the specified privilege itself.

**Removing the Privilege to One or More Roles**

One usage of the REVOKE statement is to remove roles that were assigned to a user, or a group of users, by a GRANT statement. In the case of multiple roles and/or multiple grantees, the REVOKE verb is followed by the list of roles that will be removed from the list of users specified after the FROM clause.
The optional `ADMIN OPTION FOR` clause provides the means to revoke the grantee’s “administrator” privilege, the ability to assign the same role to other users, without revoking the grantee’s privilege to the role.

Multiple roles and grantees can be processed in a single statement.

**Revoking Privileges That Were GRANTED BY**

A privilege that has been granted using the `GRANTED BY` clause is internally attributed explicitly to the grantor designated by that original `GRANT` statement. Only that user can revoke the granted privilege. Using the `GRANTED BY` clause you can revoke privileges as if you are the specified user. To revoke a privilege with `GRANTED BY`, the current user must be logged in either with full administrative privileges, or as the user designated as *grantor* by that `GRANTED BY` clause.

> Not even the owner of a role can use `GRANTED BY` unless they have administrative privileges.

The non-standard `AS` clause is supported as a synonym of the `GRANTED BY` clause to simplify migration from other database systems.

**Revoking ALL ON ALL**

The `REVOKE ALL ON ALL` statement allows a user to revoke all privileges (including roles) on all object from one or more users, roles or other database objects. It is a quick way to “clear” privileges when access to the database must be blocked for a particular user or role.

When the current user is logged in with full `administrator` privileges in the database, the `REVOKE ALL ON ALL` will remove all privileges, no matter who granted them. Otherwise, only the privileges granted by the current user are removed.

> The `GRANTED BY` clause is not supported

**Examples using REVOKE**

1. Revoking the privileges for selecting and inserting into the table (or view) `SALES`

   ```sql
   REVOKE SELECT, INSERT ON TABLE SALES
   FROM USER ALEX;
   ```

2. Revoking the privilege for selecting from the `CUSTOMER` table from the `MANAGER` and `ENGINEER` roles and from the user `IVAN`:

   ```sql
   REVOKE SELECT ON TABLE CUSTOMER
   FROM ROLE MANAGER, ROLE ENGINEER, USER IVAN;
   ```

3. Revoking from the `ADMINISTRATOR` role the privilege to grant any privileges on the `CUSTOMER` table to other users or roles:
4. Revoking the privilege for selecting from the COUNTRY table and the privilege to reference the NAME column of the COUNTRY table from any user, via the special user PUBLIC:

```sql
REVOKE SELECT, REFERENCES (NAME) ON TABLE COUNTRY
FROM PUBLIC;
```

5. Revoking the privilege for selecting form the EMPLOYEE table from the user IVAN, that was granted by the user ALEX:

```sql
REVOKE SELECT ON TABLE EMPLOYEE
FROM USER IVAN GRANTED BY ALEX;
```

6. Revoking the privilege for updating the FIRST_NAME and LAST_NAME columns of the EMPLOYEE table from the user IVAN:

```sql
REVOKE UPDATE (FIRST_NAME, LAST_NAME) ON TABLE EMPLOYEE
FROM USER IVAN;
```

7. Revoking the privilege for inserting records into the EMPLOYEE_PROJECT table from the ADD_EMP_PROJ procedure:

```sql
REVOKE INSERT ON EMPLOYEE_PROJECT
FROM PROCEDURE ADD_EMP_PROJ;
```

8. Revoking the privilege for executing the procedure ADD_EMP_PROJ from the MANAGER role:

```sql
REVOKE EXECUTE ON PROCEDURE ADD_EMP_PROJ
FROM ROLE MANAGER;
```

9. Revoking the privilege to grant the EXECUTE privilege for the function GET_BEGIN_DATE to other users from the role MANAGER:

```sql
REVOKE GRANT OPTION FOR EXECUTE
ON FUNCTION GET_BEGIN_DATE
FROM ROLE MANAGER;
```

10. Revoking the EXECUTE privilege on the package DATE_UTILS from user ALEX:

```sql
REVOKE EXECUTE ON PACKAGE DATE_UTILS
FROM USER ALEX;
```
11. Revoking the USAGE privilege on the sequence GEN_AGE from the role MANAGER:

   REVOKE USAGE ON SEQUENCE GEN_AGE
   FROM ROLE MANAGER;

12. Revoking the USAGE privilege on the sequence GEN_AGE from the trigger TR_AGE_BI:

   REVOKE USAGE ON SEQUENCE GEN_AGE
   FROM TRIGGER TR_AGE_BI;

13. Revoking the USAGE privilege on the exception E_ACCESS_DENIED from the package PKG_BILL:

   REVOKE USAGE ON EXCEPTION E_ACCESS_DENIED
   FROM PACKAGE PKG_BILL;

14. Revoking the privilege to create tables from user JOE:

   REVOKE CREATE TABLE
   FROM USER Joe;

15. Revoking the privilege to alter any procedure from user JOE:

   REVOKE ALTER ANY PROCEDURE
   FROM USER Joe;

16. Revoking the privilege to create databases from user SUPERUSER:

   REVOKE CREATE DATABASE
   FROM USER Superuser;

17. Revoking the DIRECTOR and MANAGER roles from the user IVAN:

   REVOKE DIRECTOR, MANAGER FROM USER IVAN;

18. Revoke from the user ALEX the privilege to grant the MANAGER role to other users:

   REVOKE ADMIN OPTION FOR MANAGER FROM USER ALEX;

19. Revoking all privileges (including roles) on all objects from the user IVAN:
REVOKE ALL ON ALL
FROM USER IVAN;

After this statement is executed by an administrator, the user IVAN will have no privileges whatsoever, except those granted through PUBLIC.

See also
GRANT

13.7. Mapping of Users to Objects

With Firebird now supporting multiple security databases, some new problems arise that could not occur with a single, global security database. Clusters of databases using the same security database were efficiently separated. Mappings provide the means to achieve the same efficiency when multiple databases are using their own security databases. Some cases require control for limited interaction between such clusters. For example:

- when **EXECUTE STATEMENT ON EXTERNAL DATA SOURCE** requires some data exchange between clusters
- when server-wide SYSDBA access to databases is needed from other clusters, using services.
- comparable problems that have existed on Firebird 2.1 and 2.5 for Windows, due to support for Trusted User authentication: two separate lists of users—one in the security database and another in Windows, with cases where it was necessary to relate them. An example is the demand for a **ROLE** granted to a Windows group to be assigned automatically to members of that group.

The single solution for all such cases is **mapping** the login information assigned to a user when it connects to a Firebird server to internal security objects in a database—**CURRENT_USER** and **CURRENT_ROLE**.

13.7.1. The Mapping Rule

The mapping rule consists of four pieces of information:

1. mapping scope—whether the mapping is local to the current database or whether its effect is to be global, affecting all databases in the cluster, including security databases
2. mapping name—an SQL identifier, since mappings are objects in a database, like any other
3. the object **FROM** which the mapping maps. It consists of four items:
   - The authentication source
     - plugin name or
     - the product of a mapping in another database or
     - use of server-wide authentication or
     - any method
   - The name of the database where authentication succeeded
The name of the object from which mapping is performed
- The type of that name—username, role, or OS group—depending upon the plugin that added that name during authentication.

Any item is accepted but only type is required.

4. the object TO which the mapping maps. It consists of two items:
- The name of the object TO which mapping is performed
- The type, for which only USER or ROLE is valid

### 13.7.2. CREATE MAPPING

**Used for**
Creating a mapping of a security object

**Available in**
DSQL

**Syntax**

```
CREATE [GLOBAL] MAPPING name
USING
{ PLUGIN plugin_name [IN database]
  | ANY PLUGIN [IN database | SERVERWIDE]
  | MAPPING [IN database] | '*' [IN database] }
FROM {ANY type | type from_name}
TO {USER | ROLE} [to_name]
```

**Table 228. CREATE MAPPING Statement Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Mapping name The maximum length is 31 characters. Must be unique among all mapping names in the context (local or GLOBAL).</td>
</tr>
<tr>
<td>plugin_name</td>
<td>Authentication plugin name</td>
</tr>
<tr>
<td>database</td>
<td>Name of the database that authenticated against</td>
</tr>
<tr>
<td>type</td>
<td>The type of object to be mapped. Possible types are plugin-specific.</td>
</tr>
<tr>
<td>from_name</td>
<td>The name of the object to be mapped</td>
</tr>
<tr>
<td>to_name</td>
<td>The name of the user or role to map to</td>
</tr>
</tbody>
</table>

The CREATE MAPPING statement creates a mapping of security objects (e.g. users, groups, roles) of one or more authentication plugins to internal security objects - CURRENT_USER and CURRENT_ROLE.

If the GLOBAL clause is present, then the mapping will be applied not only for the current database, but for all databases in the same cluster, including security databases.

⚠️ There can be global and local mappings with the same name. They are distinct
Global mapping works best if a Firebird 3.0 or higher version database is used as the security database. If you plan to use another database for this purpose — using your own provider, for example — then you should create a table in it named RDB$MAP, with the same structure as RDB$MAP in a Firebird 3.0 database and with SYSDBA-only write access.

The USING clause describes the mapping source. It has a very complex set of options:

- an explicit plugin name (PLUGIN plugin_name) means it applies only for that plugin
- it can use any available plugin (ANY PLUGIN); although not if the source is the product of a previous mapping
- it can be made to work only with server-wide plugins (SERVERWIDE)
- it can be made to work only with previous mapping results (MAPPING)
- you can omit to use of a specific method by using the asterisk (*) argument
- it can specify the name of the database that defined the mapping for the FROM object (IN database)

This argument is not valid for mapping server-wide authentication.

The FROM clause describes the object to map. The FROM clause has a mandatory argument, the type of the object named. It has the following options:

- When mapping names from plugins, type is defined by the plugin
- When mapping the product of a previous mapping, type can be only USER or ROLE
- If an explicit from_name is provided, it will be taken into account by this mapping
- Use the ANY keyword to work with any name of the given type.

The TO clause specifies the user or role that is the result of the mapping. The to_name is optional. If it is not specified, then the original name of the mapped object will be used.

For roles, the role defined by a mapping rule is only applied when the user does not explicitly specify a role on connect. The mapped role can be assumed later in the session using SET TRUSTED ROLE, even when the mapped role is not explicitly granted to the user.

**Who Can Create a Mapping**

The CREATE MAPPING statement can be executed by:

- Administrators
- The database owner — if the mapping is local

**CREATE MAPPING examples**

1. Enable use of Windows trusted authentication in all databases that use the current security
database:

CREATE GLOBAL MAPPING TRUSTED_AUTH
    USING PLUGIN WIN_SSPI
    FROM ANY USER
    TO USER;

2. Enable RDB$ADMIN access for windows admins in the current database:

CREATE MAPPING WIN_ADMINS
    USING PLUGIN WIN_SSPI
    FROM Predefined_Group
    DOMAIN_ANY_RID_ADMINS
    TO ROLE RDB$ADMIN;

The group DOMAIN_ANY_RID_ADMINS does not exist in Windows, but such a name would be added by the Win_Sspi plugin to provide exact backwards compatibility.

3. Enable a particular user from another database to access the current database with another name:

CREATE MAPPING FROM_RT
    USING PLUGIN SRP IN "rt"
    FROM USER U1 TO USER U2;

Database names or aliases will need to be enclosed in double quotes on operating systems that have case-sensitive file names.

4. Enable the server's SYSDBA (from the main security database) to access the current database. (Assume that the database is using a non-default security database):

CREATE MAPPING DEF_SYSDBA
    USING PLUGIN SRP IN "security.db"
    FROM USER SYSDBA
    TO USER;

5. Ensure users who logged in using the legacy authentication plugin do not have too many privileges:

CREATE MAPPING LEGACY_2_GUEST
    USING PLUGIN legacy_auth
    FROM ANY USER
    TO USER GUEST;
13.7.3. ALTER MAPPING

Used for
Altering a mapping of a security object

Available in
DSQL

Syntax

```
ALTER [GLOBAL] MAPPING name
  USING
     { PLUGIN plugin_name [IN database]
       | ANY PLUGIN [IN database | SERVERWIDE]
       | MAPPING [IN database] | '*' [IN database] }
  FROM {ANY type | type from_name}
  TO {USER | ROLE} [to_name]
```

For details on the options, see CREATE MAPPING.

The ALTER MAPPING statement allows you to modify any of the existing mapping options, but a local mapping cannot be changed to GLOBAL or vice versa.

⚠️ Global and local mappings of the same name are different objects.

Who Can Alter a Mapping

The ALTER MAPPING statement can be executed by:

- Administrators
- The database owner — if the mapping is local

ALTER MAPPING examples

Alter mapping

```
ALTER MAPPING FROM_RT
  USING PLUGIN SRP IN "rt"
  FROM USER U1 TO USER U3;
```

See also
CREATE MAPPING, CREATE OR ALTER MAPPING, DROP MAPPING
13.7.4. CREATE OR ALTER MAPPING

Used for
Creating a new or altering an existing mapping of a security object

Available in
DSQL

Syntax

```sql
CREATE OR ALTER [GLOBAL] MAPPING name
  USING
  { PLUGIN plugin_name [IN database]
  | ANY PLUGIN [IN database | SERVERWIDE]
  | MAPPING [IN database] | '*' [IN database] } 
FROM {ANY type | type from_name}
TO {USER | ROLE} [to_name]
```

For details on the options, see CREATE MAPPING.

The CREATE OR ALTER MAPPING statement creates a new or modifies an existing mapping.

⚠️ Global and local mappings of the same name are different objects.

CREATE OR ALTER MAPPING examples

Creating or altering a mapping

```sql
CREATE OR ALTER MAPPING FROM_RT
  USING PLUGIN SRP IN "rt"
  FROM USER U1 TO USER U4;
```

See also
CREATE MAPPING, ALTER MAPPING, DROP MAPPING

13.7.5. DROP MAPPING

Used for
Dropping (removing) a mapping of a security object

Available in
DSQL

Syntax

```sql
DROP [GLOBAL] MAPPING name
```

Table 229. DROP MAPPING Statement Parameter
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Mapping name</td>
</tr>
</tbody>
</table>

The **DROP MAPPING** statement removes an existing mapping. If **GLOBAL** is specified, then a global mapping will be removed.

⚠️ Global and local mappings of the same name are different objects.

**Who Can Drop a Mapping**

The **DROP MAPPING** statement can be executed by:

- Administrators
- The database owner — if the mapping is local

**DROP MAPPING examples**

**Alter mapping**

```sql
DROP MAPPING FROM_RT;
```

*See also*

**CREATE MAPPING**

### 13.8. Database Encryption

Firebird provides a plugin mechanism to encrypt the data stored in the database. This mechanism does not encrypt the entire database, but only data pages, index pages, and blob pages.

In order to make database encryption possible, you need to obtain or write a database encryption plugin.

Out of the box, Firebird does not include a database encryption plugin.

The encryption plugin example in `examples/dbcrypt` does not perform real encryption, it is only intended as an example how such a plugin can be written.

On Linux, an example plugin named `libDbCrypt_example.so` can be found in `plugins/`.

The main problem with database encryption is how to store the secret key. Firebird provides support for transferring the key from the client, but this does not mean that storing the key on the client is the best way; it is just one of the possible alternatives. However, keeping encryption keys on the same disk as the database is an insecure option.

For efficient separation of encryption and key access, the database encryption plugin data is divided into two parts, the encryption itself and the holder of the secret key. This can be an efficient approach when you want to use some good encryption algorithm, but you have your own custom
method of storing the keys.

Once you have decided on the plugin and key holder, you can perform the encryption.

### 13.8.1. Encrypting a Database

**Syntax**

```
ALTER {DATABASE | SCHEMA}
    ENCRYPT WITH plugin_name [KEY key_name]
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plugin_name</td>
<td>The name of the encryption plugin</td>
</tr>
<tr>
<td>key_name</td>
<td>The name of the encryption key</td>
</tr>
</tbody>
</table>

Encrypts the database using the specified encryption plugin. Encryption starts immediately after this statement completes, and will be performed in the background. Normal operations of the database are not disturbed during encryption.

The optional `KEY` clause specifies the name of the key for the encryption plugin. The plugin decides what to do with this key name.

The encryption process can be monitored using the `MON$CRYPT_PAGE` field in the `MON$DATABASE` virtual table, or viewed in the database header page using `gstat -e`. `gstat -h` will also provide limited information about the encryption status.

For example, the following query will display the progress of the encryption process as a percentage.

```
select MON$CRYPT_PAGE * 100 / MON$PAGES
    from MON$DATABASE;
```

SCHEMA is currently a synonym for DATABASE; this may change in a future version, so we recommend to always use DATABASE

*See also*

Decrypting a Database, ALTER DATABASE

### 13.8.2. Decrypting a Database

**Syntax**

```
ALTER {DATABASE | SCHEMA} DECRYPT
```
Decrypts the database using the configured plugin and key. Decryption starts immediately after this statement completes, and will be performed in the background. Normal operations of the database are not disturbed during decryption.

|  | SCHEMA is currently a synonym for DATABASE; this may change in a future version, so we recommend to always use DATABASE |

*See also*

Encrypting a Database, ALTER DATABASE
Chapter 14. Management Statements

Since Firebird 3.0 a new class of DSQL statement has emerged in Firebird’s SQL lexicon, usually for administering aspects of the client/server environment. Typically, such statements start with the verb `SET`.

The `isql` tool also has a collection of `SET` commands. Those commands are not part of Firebird’s SQL lexicon. For information on `isql` SET commands, see `Isql Set Commands` in `Firebird Interactive SQL Utility`.

Most of the management statements affect the current connection (attachment, or “session”) only, and do not require any authorization over and above the login privileges of the current user without elevated privileges.

### 14.1. Changing the Current Role

#### 14.1.1. SET ROLE

*Used for*

Changing the role of the current session

*Available in*

DSQL

*Syntax*

```
SET ROLE {role_name | NONE}
```

*Table 231. SET ROLE Statement Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>role_name</td>
<td>The name of the role to apply</td>
</tr>
</tbody>
</table>

The `SET ROLE` statement allows a user to assume a different role; it sets the CURRENT_ROLE context variable to `role_name`, if that role has been granted to the CURRENT_USER. For this session, the user receives the privileges granted by that role. Any rights granted to the previous role are removed from the session. Use `NONE` instead of `role_name` to clear the CURRENT_ROLE.

When the specified role does not exist or has not been explicitly granted to the user, the error “Role `role_name` is invalid or unavailable” is raised.

*SET ROLE Examples*

1. Change the current role to MANAGER

   ```
   SET ROLE manager;
   select current_role from rdb$database;
   ```
2. Clear the current role

```
SET ROLE NONE;
select current_role from rdb$database;

ROLE
=======================
NONE
```

See also

SET TRUSTED ROLE, GRANT

### 14.1.2. SET TRUSTED ROLE

**Used for**

Changes role of the current session to the trusted role

**Available in**

DSQL

**Syntax**

```
SET TRUSTED ROLE
```

The `SET TRUSTED ROLE` statement makes it possible to assume the role assigned to the user through a mapping rule (see **Mapping of Users to Objects**). The role assigned through a mapping rule is assumed automatically on connect, if the user hasn’t specified an explicit role. The `SET TRUSTED ROLE` statement makes it possible to assume the mapped (or “trusted”) role at a later time, or to assume it again after the current role was changed using `SET ROLE`.

A trusted role is not a specific type of role, but can be any role that was created using `CREATE ROLE`, or a predefined system role such as `RDB$ADMIN`. An attachment (session) has a trusted role when the **security objects mapping subsystem** finds a match between the authentication result passed from the plugin and a local or global mapping to a role for the current database. The role may be one that is not granted explicitly to that user.

When a session has no trusted role, executing `SET TRUSTED ROLE` will raise error "Your attachment has no trusted role".

While the `CURRENT_ROLE` can be changed using `SET ROLE`, it is not always possible to revert to a trusted role using the same command, because `SET ROLE` checks if the role has been granted to the user. With `SET TRUSTED ROLE`, the trusted role can be
assumed again even when SET ROLE fails.

**SET TRUSTED ROLE Examples**

1. Assuming a mapping rule that assigns the role ROLE1 to a user ALEX:

   ```sql
   CONNECT 'employee' USER ALEX PASSWORD 'password';
   SELECT CURRENT_ROLE FROM RDB$DATABASE;
   
   ROLE
   ================
   ROLE1
   
   SET ROLE ROLE2;
   SELECT CURRENT_ROLE FROM RDB$DATABASE;
   
   ROLE
   ================
   ROLE2
   
   SET TRUSTED ROLE;
   SELECT CURRENT_ROLE FROM RDB$DATABASE;
   
   ROLE
   ================
   ROLE1
   
   See also
   
   **SET ROLE, Mapping of Users to Objects**
Appendix A: Supplementary Information

In this Appendix are topics that developers may wish to refer to, to enhance understanding of features or changes.

The RDB$VALID_BLR Field

The field RDB$VALID_BLR was added to the system tables RDB$PROCEDURES and RDB$TRIGGERS in Firebird 2.1. Its purpose is to signal possible invalidation of a PSQL module after alteration of a domain or table column on which the module depends. RDB$VALID_BLR is set to 0 for any procedure or trigger whose code is made invalid by such a change.

How Invalidation Works

In triggers and procedures, dependencies arise on the definitions of table columns accessed and also on any parameter or variable that has been defined in the module using the TYPE OF clause.

After the engine has altered any domain, including the implicit domains created internally behind column definitions and output parameters, the engine internally recompiles all of its dependencies.

In v2.x these comprise procedures and triggers but not blocks coded in DML statements for run-time execution with EXECUTE BLOCK. Firebird 3 will encompass more module types (stored functions, packages).

Any module that fails to recompile because of an incompatibility arising from a domain change is marked as invalid (“invalidated” by setting the RDB$VALID_BLR in its system record (in RDB$PROCEDURES or RDB$TRIGGERS, as appropriate) to zero.

Revalidation (setting RDB$VALID_BLR to 1) occurs when

1. the domain is altered again and the new definition is compatible with the previously invalidated module definition; OR
2. the previously invalidated module is altered to match the new domain definition

The following query will find the modules that depend on a specific domain and report the state of their RDB$VALID_BLR fields:

```
SELECT * FROM (
  SELECT
    'Procedure',
    rdb$procedure_name,
    rdb$valid_blr
  FROM rdb$procedures
  UNION ALL
  SELECT
    'Trigger',
    rdb$trigger_name,
    rdb$valid_blr
  FROM rdb$triggers
)
```
FROM rdb$triggers
) (type, name, valid)
WHERE EXISTS
(SELECT * from rdb$dependencies
WHERE rdb$dependent_name = name
   AND rdb$depended_on_name = 'MYDOMAIN')

/* Replace MYDOMAIN with the actual domain name.
   Use all-caps if the domain was created
   case-insensitively. Otherwise, use the exact
   capitalisation. */

The following query will find the modules that depend on a specific table column and report the
state of their RDB$VALID_BLR fields:

SELECT * FROM (SELECT
   'Procedure',
   rdb$procedure_name,
   rdb$valid_blr
FROM rdb$procedures
UNION ALL
SELECT
   'Trigger',
   rdb$trigger_name,
   rdb$valid_blr
FROM rdb$triggers) (type, name, valid)
WHERE EXISTS
(SELECT *
FROM rdb$dependencies
WHERE rdb$dependent_name = name
   AND rdb$depended_on_name = 'MYTABLE'
   AND rdb$field_name = 'MYCOLUMN')

All PSQL invalidations caused by domain/column changes are reflected in the
RDB$VALID_BLR field. However, other kinds of changes, such as the number of input
or output parameters, called routines and so on, do not affect the validation field
even though they potentially invalidate the module. A typical such scenario might
be one of the following:

1. A procedure (B) is defined, that calls another procedure (A) and reads output
parameters from it. In this case, a dependency is registered in
RDB$DEPENDENCIES. Subsequently, the called procedure (A) is altered to change or
remove one or more of those output parameters. The ALTER PROCEDURE A
statement will fail with an error when commit is attempted.

2. A procedure (B) calls procedure A, supplying values for its input parameters.
No dependency is registered in RDB$DEPENDENCIES. Subsequent modification of
the input parameters in procedure A will be allowed. Failure will occur at run-
time, when B calls A with the mismatched input parameter set.

**Other Notes**

- For PSQL modules inherited from earlier Firebird versions (including a number of system triggers, even if the database was created under Firebird 2.1 or higher), RDB$VALID_BLR is NULL. This does not imply that their BLR is invalid.
- The `isql` commands `SHOW PROCEDURES` and `SHOW TRIGGERS` display an asterisk in the RDB$VALID_BLR column for any module for which the value is zero (i.e., invalid). However, `SHOW PROCEDURE <procname>` and `SHOW TRIGGER <trigname>`, which display individual PSQL modules, do not signal invalid BLR at all.

**A Note on Equality**

This note about equality and inequality operators applies everywhere in Firebird’s SQL language.

The “=” operator, which is explicitly used in many conditions, only matches values to values. According to the SQL standard, NULL is not a value and hence two NULLs are neither equal nor unequal to one another. If you need NULLs to match each other in a condition, use the IS NOT DISTINCT FROM operator. This operator returns true if the operands have the same value or if they are both NULL.

```sql
select *
from A join B
on A.id is not distinct from B.code
```

Likewise, in cases where you want to test against NULL for a condition of *inequality*, use IS DISTINCT FROM, not “<>”. If you want NULL to be considered different from any value and two NULLs to be considered equal:

```sql
select *
from A join B
on A.id is distinct from B.code
```
Appendix B: Exception Codes and Messages

This appendix includes:

- SQLSTATE Error Codes and Descriptions
- GDSCODE Error Codes, SQLCODEs and Descriptions

Custom Exceptions

Firebird DDL provides a simple syntax for creating custom exceptions for use in PSQL modules, with message text of up to 1,021 characters. For more information, see `CREATE EXCEPTION` in `DDL Statements` and, for usage, the statement `EXCEPTION` in `PSQL Statements`.

The Firebird SQLCODE error codes do not correlate with the standards-compliant SQLSTATE codes. SQLCODE has been used for many years and should be considered as deprecated now. Support for SQLCODE is likely to be dropped in a future version.

SQLSTATE Error Codes and Descriptions

This table provides the error codes and message texts for the SQLSTATE context variables.

The structure of an SQLSTATE error code is five characters comprising the SQL error class (2 characters) and the SQL subclass (3 characters).

Table 232. SQLSTATE Codes and Message Texts

<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Mapped Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCLASS 00 (Success)</td>
<td>00000 Success</td>
</tr>
<tr>
<td>SQLCLASS 01 (Warning)</td>
<td>01000 General warning</td>
</tr>
<tr>
<td></td>
<td>01001 Cursor operation conflict</td>
</tr>
<tr>
<td></td>
<td>01002 Disconnect error</td>
</tr>
<tr>
<td></td>
<td>01003 NULL value eliminated in set function</td>
</tr>
<tr>
<td></td>
<td>01004 String data, right-truncated</td>
</tr>
<tr>
<td></td>
<td>01005 Insufficient item descriptor areas</td>
</tr>
<tr>
<td></td>
<td>01006 Privilege not revoked</td>
</tr>
<tr>
<td></td>
<td>01007 Privilege not granted</td>
</tr>
<tr>
<td></td>
<td>01008 Implicit zero-bit padding</td>
</tr>
<tr>
<td></td>
<td>01100 Statement reset to unprepared</td>
</tr>
<tr>
<td></td>
<td>01101 Ongoing transaction has been committed</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>Mapped Message</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
</tr>
<tr>
<td>01102</td>
<td>Ongoing transaction has been rolled back</td>
</tr>
<tr>
<td>02000</td>
<td>No data found or no rows affected</td>
</tr>
<tr>
<td>07000</td>
<td>Dynamic SQL error</td>
</tr>
<tr>
<td>07001</td>
<td>Wrong number of input parameters</td>
</tr>
<tr>
<td>07002</td>
<td>Wrong number of output parameters</td>
</tr>
<tr>
<td>07003</td>
<td>Cursor specification cannot be executed</td>
</tr>
<tr>
<td>07004</td>
<td>USING clause required for dynamic parameters</td>
</tr>
<tr>
<td>07005</td>
<td>Prepared statement not a cursor-specification</td>
</tr>
<tr>
<td>07006</td>
<td>Restricted data type attribute violation</td>
</tr>
<tr>
<td>07007</td>
<td>USING clause required for result fields</td>
</tr>
<tr>
<td>07008</td>
<td>Invalid descriptor count</td>
</tr>
<tr>
<td>07009</td>
<td>Invalid descriptor index</td>
</tr>
<tr>
<td>08001</td>
<td>Client unable to establish connection</td>
</tr>
<tr>
<td>08002</td>
<td>Connection name in use</td>
</tr>
<tr>
<td>08003</td>
<td>Connection does not exist</td>
</tr>
<tr>
<td>08004</td>
<td>Server rejected the connection</td>
</tr>
<tr>
<td>08006</td>
<td>Connection failure</td>
</tr>
<tr>
<td>08007</td>
<td>Transaction resolution unknown</td>
</tr>
<tr>
<td>0A000</td>
<td>Feature Not Supported</td>
</tr>
<tr>
<td>0B000</td>
<td>Invalid transaction initiation</td>
</tr>
<tr>
<td>0L000</td>
<td>Invalid grantor</td>
</tr>
<tr>
<td>0P000</td>
<td>Invalid role specification</td>
</tr>
<tr>
<td>0U000</td>
<td>Attempt to assign to non-updatable column</td>
</tr>
<tr>
<td>0V000</td>
<td>Attempt to assign to Ordering column</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>Mapped Message</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>SQLCLASS 20 (Case Not Found For Case Statement)</td>
<td></td>
</tr>
<tr>
<td>20000</td>
<td>Case not found for case statement</td>
</tr>
<tr>
<td>SQLCLASS 21 (Cardinality Violation)</td>
<td></td>
</tr>
<tr>
<td>21000</td>
<td>Cardinality violation</td>
</tr>
<tr>
<td>21S01</td>
<td>Insert value list does not match column list</td>
</tr>
<tr>
<td>21S02</td>
<td>Degree of derived table does not match column list</td>
</tr>
<tr>
<td>SQLCLASS 22 (Data Exception)</td>
<td></td>
</tr>
<tr>
<td>22000</td>
<td>Data exception</td>
</tr>
<tr>
<td>22001</td>
<td>String data, right truncation</td>
</tr>
<tr>
<td>22002</td>
<td>Null value, no indicator parameter</td>
</tr>
<tr>
<td>22003</td>
<td>Numeric value out of range</td>
</tr>
<tr>
<td>22004</td>
<td>Null value not allowed</td>
</tr>
<tr>
<td>22005</td>
<td>Error in assignment</td>
</tr>
<tr>
<td>22006</td>
<td>Null value in field reference</td>
</tr>
<tr>
<td>22007</td>
<td>Invalid datetime format</td>
</tr>
<tr>
<td>22008</td>
<td>Datetime field overflow</td>
</tr>
<tr>
<td>22009</td>
<td>Invalid time zone displacement value</td>
</tr>
<tr>
<td>2200A</td>
<td>Null value in reference target</td>
</tr>
<tr>
<td>2200B</td>
<td>Escape character conflict</td>
</tr>
<tr>
<td>2200C</td>
<td>Invalid use of escape character</td>
</tr>
<tr>
<td>2200D</td>
<td>Invalid escape octet</td>
</tr>
<tr>
<td>2200E</td>
<td>Null value in array target</td>
</tr>
<tr>
<td>2200F</td>
<td>Zero-length character string</td>
</tr>
<tr>
<td>2200G</td>
<td>Most specific type mismatch</td>
</tr>
<tr>
<td>22010</td>
<td>Invalid indicator parameter value</td>
</tr>
<tr>
<td>22011</td>
<td>Substring error</td>
</tr>
<tr>
<td>22012</td>
<td>Division by zero</td>
</tr>
<tr>
<td>22014</td>
<td>Invalid update value</td>
</tr>
<tr>
<td>22015</td>
<td>Interval field overflow</td>
</tr>
<tr>
<td>22018</td>
<td>Invalid character value for cast</td>
</tr>
<tr>
<td>22019</td>
<td>Invalid escape character</td>
</tr>
<tr>
<td>2201B</td>
<td>Invalid regular expression</td>
</tr>
<tr>
<td>2201C</td>
<td>Null row not permitted in table</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>Mapped Message</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------------</td>
</tr>
<tr>
<td>22012</td>
<td>Division by zero</td>
</tr>
<tr>
<td>22020</td>
<td>Invalid limit value</td>
</tr>
<tr>
<td>22021</td>
<td>Character not in repertoire</td>
</tr>
<tr>
<td>22022</td>
<td>Indicator overflow</td>
</tr>
<tr>
<td>22023</td>
<td>Invalid parameter value</td>
</tr>
<tr>
<td>22024</td>
<td>Character string not properly terminated</td>
</tr>
<tr>
<td>22025</td>
<td>Invalid escape sequence</td>
</tr>
<tr>
<td>22026</td>
<td>String data, length mismatch</td>
</tr>
<tr>
<td>22027</td>
<td>Trim error</td>
</tr>
<tr>
<td>22028</td>
<td>Row already exists</td>
</tr>
<tr>
<td>2202D</td>
<td>Null instance used in mutator function</td>
</tr>
<tr>
<td>2202E</td>
<td>Array element error</td>
</tr>
<tr>
<td>2202F</td>
<td>Array data, right truncation</td>
</tr>
<tr>
<td>23000</td>
<td>Integrity constraint violation</td>
</tr>
<tr>
<td>24000</td>
<td>Invalid cursor state</td>
</tr>
<tr>
<td>24504</td>
<td>The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row</td>
</tr>
<tr>
<td>25000</td>
<td>Invalid transaction state</td>
</tr>
<tr>
<td>25S01</td>
<td>Transaction state</td>
</tr>
<tr>
<td>25S02</td>
<td>Transaction is still active</td>
</tr>
<tr>
<td>25S03</td>
<td>Transaction is rolled back</td>
</tr>
<tr>
<td>26000</td>
<td>Invalid SQL statement name</td>
</tr>
<tr>
<td>27000</td>
<td>Triggered data change violation</td>
</tr>
<tr>
<td>28000</td>
<td>Invalid authorization specification</td>
</tr>
<tr>
<td>2B000</td>
<td>Dependent privilege descriptors still exist</td>
</tr>
<tr>
<td>2C000</td>
<td>Invalid character set name</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>Mapped Message</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>SQLCLASS 2D</td>
<td>Invalid Transaction Termination</td>
</tr>
<tr>
<td>2D000</td>
<td>Invalid transaction termination</td>
</tr>
<tr>
<td>SQLCLASS 2E</td>
<td>Invalid Connection Name</td>
</tr>
<tr>
<td>2E000</td>
<td>Invalid connection name</td>
</tr>
<tr>
<td>SQLCLASS 2F</td>
<td>SQL Routine Exception</td>
</tr>
<tr>
<td>2F000</td>
<td>SQL routine exception</td>
</tr>
<tr>
<td>2F002</td>
<td>Modifying SQL-data not permitted</td>
</tr>
<tr>
<td>2F003</td>
<td>Prohibited SQL-statement attempted</td>
</tr>
<tr>
<td>2F004</td>
<td>Reading SQL-data not permitted</td>
</tr>
<tr>
<td>2F005</td>
<td>Function executed no return statement</td>
</tr>
<tr>
<td>SQLCLASS 33</td>
<td>Invalid SQL Descriptor Name</td>
</tr>
<tr>
<td>33000</td>
<td>Invalid SQL descriptor name</td>
</tr>
<tr>
<td>SQLCLASS 34</td>
<td>Invalid Cursor Name</td>
</tr>
<tr>
<td>34000</td>
<td>Invalid cursor name</td>
</tr>
<tr>
<td>SQLCLASS 35</td>
<td>Invalid Condition Number</td>
</tr>
<tr>
<td>35000</td>
<td>Invalid condition number</td>
</tr>
<tr>
<td>SQLCLASS 36</td>
<td>Cursor Sensitivity Exception</td>
</tr>
<tr>
<td>36001</td>
<td>Request rejected</td>
</tr>
<tr>
<td>36002</td>
<td>Request failed</td>
</tr>
<tr>
<td>SQLCLASS 37</td>
<td>Invalid Identifier</td>
</tr>
<tr>
<td>37000</td>
<td>Invalid identifier</td>
</tr>
<tr>
<td>37001</td>
<td>Identifier too long</td>
</tr>
<tr>
<td>SQLCLASS 38</td>
<td>External Routine Exception</td>
</tr>
<tr>
<td>38000</td>
<td>External routine exception</td>
</tr>
<tr>
<td>SQLCLASS 39</td>
<td>External Routine Invocation Exception</td>
</tr>
<tr>
<td>39000</td>
<td>External routine invocation exception</td>
</tr>
<tr>
<td>SQLCLASS 3B</td>
<td>Invalid Save Point</td>
</tr>
<tr>
<td>3B000</td>
<td>Invalid save point</td>
</tr>
<tr>
<td>SQLCLASS 3C</td>
<td>Ambiguous Cursor Name</td>
</tr>
<tr>
<td>3C000</td>
<td>Ambiguous cursor name</td>
</tr>
<tr>
<td>SQLCLASS 3D</td>
<td>Invalid Catalog Name</td>
</tr>
<tr>
<td>3D000</td>
<td>Invalid catalog name</td>
</tr>
<tr>
<td>3D001</td>
<td>Catalog name not found</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>Mapped Message</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
</tr>
<tr>
<td>SQLCLASS 3F (Invalid Schema Name)</td>
<td></td>
</tr>
<tr>
<td>3F000</td>
<td>Invalid schema name</td>
</tr>
<tr>
<td>SQLCLASS 40 (Transaction Rollback)</td>
<td></td>
</tr>
<tr>
<td>40000</td>
<td>Ongoing transaction has been rolled back</td>
</tr>
<tr>
<td>40001</td>
<td>Serialization failure</td>
</tr>
<tr>
<td>40002</td>
<td>Transaction integrity constraint violation</td>
</tr>
<tr>
<td>40003</td>
<td>Statement completion unknown</td>
</tr>
<tr>
<td>SQLCLASS 42 (Syntax Error or Access Violation)</td>
<td></td>
</tr>
<tr>
<td>42000</td>
<td>Syntax error or access violation</td>
</tr>
<tr>
<td>42702</td>
<td>Ambiguous column reference</td>
</tr>
<tr>
<td>42725</td>
<td>Ambiguous function reference</td>
</tr>
<tr>
<td>42818</td>
<td>The operands of an operator or function are not compatible</td>
</tr>
<tr>
<td>42S01</td>
<td>Base table or view already exists</td>
</tr>
<tr>
<td>42S02</td>
<td>Base table or view not found</td>
</tr>
<tr>
<td>42S11</td>
<td>Index already exists</td>
</tr>
<tr>
<td>42S12</td>
<td>Index not found</td>
</tr>
<tr>
<td>42S21</td>
<td>Column already exists</td>
</tr>
<tr>
<td>42S22</td>
<td>Column not found</td>
</tr>
<tr>
<td>SQLCLASS 44 (With Check Option Violation)</td>
<td></td>
</tr>
<tr>
<td>44000</td>
<td>WITH CHECK OPTION Violation</td>
</tr>
<tr>
<td>SQLCLASS 45 (Unhandled User-defined Exception)</td>
<td></td>
</tr>
<tr>
<td>45000</td>
<td>Unhandled user-defined exception</td>
</tr>
<tr>
<td>SQLCLASS 54 (Program Limit Exceeded)</td>
<td></td>
</tr>
<tr>
<td>54000</td>
<td>Program limit exceeded</td>
</tr>
<tr>
<td>54001</td>
<td>Statement too complex</td>
</tr>
<tr>
<td>54011</td>
<td>Too many columns</td>
</tr>
<tr>
<td>54023</td>
<td>Too many arguments</td>
</tr>
<tr>
<td>SQLCLASS HY (CLI-specific Condition)</td>
<td></td>
</tr>
<tr>
<td>HY000</td>
<td>CLI-specific condition</td>
</tr>
<tr>
<td>HY001</td>
<td>Memory allocation error</td>
</tr>
<tr>
<td>HY003</td>
<td>Invalid data type in application descriptor</td>
</tr>
<tr>
<td>HY004</td>
<td>Invalid data type</td>
</tr>
<tr>
<td>HY007</td>
<td>Associated statement is not prepared</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>Mapped Message</td>
</tr>
<tr>
<td>----------</td>
<td>---------------</td>
</tr>
<tr>
<td>HY008</td>
<td>Operation canceled</td>
</tr>
<tr>
<td>HY009</td>
<td>Invalid use of null pointer</td>
</tr>
<tr>
<td>HY010</td>
<td>Function sequence error</td>
</tr>
<tr>
<td>HY011</td>
<td>Attribute cannot be set now</td>
</tr>
<tr>
<td>HY012</td>
<td>Invalid transaction operation code</td>
</tr>
<tr>
<td>HY013</td>
<td>Memory management error</td>
</tr>
<tr>
<td>HY014</td>
<td>Limit on the number of handles exceeded</td>
</tr>
<tr>
<td>HY015</td>
<td>No cursor name available</td>
</tr>
<tr>
<td>HY016</td>
<td>Cannot modify an implementation row descriptor</td>
</tr>
<tr>
<td>HY017</td>
<td>Invalid use of an automatically allocated descriptor handle</td>
</tr>
<tr>
<td>HY018</td>
<td>Server declined the cancellation request</td>
</tr>
<tr>
<td>HY019</td>
<td>Non-string data cannot be sent in pieces</td>
</tr>
<tr>
<td>HY020</td>
<td>Attempt to concatenate a null value</td>
</tr>
<tr>
<td>HY021</td>
<td>Inconsistent descriptor information</td>
</tr>
<tr>
<td>HY024</td>
<td>Invalid attribute value</td>
</tr>
<tr>
<td>HY055</td>
<td>Non-string data cannot be used with string routine</td>
</tr>
<tr>
<td>HY090</td>
<td>Invalid string length or buffer length</td>
</tr>
<tr>
<td>HY091</td>
<td>Invalid descriptor field identifier</td>
</tr>
<tr>
<td>HY092</td>
<td>Invalid attribute identifier</td>
</tr>
<tr>
<td>HY095</td>
<td>Invalid Function ID specified</td>
</tr>
<tr>
<td>HY096</td>
<td>Invalid information type</td>
</tr>
<tr>
<td>HY097</td>
<td>Column type out of range</td>
</tr>
<tr>
<td>HY098</td>
<td>Scope out of range</td>
</tr>
<tr>
<td>HY099</td>
<td>Nullable type out of range</td>
</tr>
<tr>
<td>HY100</td>
<td>Uniqueness option type out of range</td>
</tr>
<tr>
<td>HY101</td>
<td>Accuracy option type out of range</td>
</tr>
<tr>
<td>HY103</td>
<td>Invalid retrieval code</td>
</tr>
<tr>
<td>HY104</td>
<td>Invalid Length/Precision value</td>
</tr>
<tr>
<td>HY105</td>
<td>Invalid parameter type</td>
</tr>
<tr>
<td>HY106</td>
<td>Invalid fetch orientation</td>
</tr>
<tr>
<td>HY107</td>
<td>Row value out of range</td>
</tr>
<tr>
<td>HY109</td>
<td>Invalid cursor position</td>
</tr>
<tr>
<td>HY110</td>
<td>Invalid driver completion</td>
</tr>
</tbody>
</table>
SQLSTATE | Mapped Message
--- | ---
HY111  | Invalid bookmark value
HYC00  | Optional feature not implemented
HYT00  | Timeout expired
HYT01  | Connection timeout expired

SQLCLASS XX (Internal Error)

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>XX000</td>
<td>Internal error</td>
</tr>
<tr>
<td>XX001</td>
<td>Data corrupted</td>
</tr>
<tr>
<td>XX002</td>
<td>Index corrupted</td>
</tr>
</tbody>
</table>

**SQLCODE and GDSCODE Error Codes and Descriptions**

The table provides the SQLCODE groupings, the numeric and symbolic values for the GDSCODE errors and the message texts.

*SQLCODE has been used for many years and should be considered as deprecated now. Support for SQLCODE is likely to be dropped in a future version.*

*Table 233. SQLCODE and GDSCODE Error Codes and Message Texts*

<table>
<thead>
<tr>
<th>SQLCODE</th>
<th>GDSCODE</th>
<th>Symbol</th>
<th>Message Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>335544802</td>
<td>dialect_reset_warning</td>
<td>Database dialect being changed from 3 to 1</td>
</tr>
<tr>
<td>301</td>
<td>335544808</td>
<td>dtype_renamed</td>
<td>DATE data type is now called TIMESTAMP</td>
</tr>
<tr>
<td>301</td>
<td>336003076</td>
<td>dsql_dialect_warning_expr</td>
<td>Use of @1 expression that returns different results in dialect 1 and dialect 3</td>
</tr>
<tr>
<td>301</td>
<td>336003080</td>
<td>dsql_warning_number_ambiguous</td>
<td>WARNING: Numeric literal @1 is interpreted as a floating-point value in SQL dialect 1, but as an exact numeric value in SQL dialect 3.</td>
</tr>
<tr>
<td>301</td>
<td>336003081</td>
<td>dsql_warning_number_ambiguous1</td>
<td>WARNING: NUMERIC and DECIMAL fields with precision 10 or greater are stored</td>
</tr>
<tr>
<td>301</td>
<td>336003082</td>
<td>dsql_warn_precision_ambiguous</td>
<td></td>
</tr>
<tr>
<td>301</td>
<td>336003083</td>
<td>dsql_warn_precision_ambiguous1</td>
<td>as approximate floating-point values in SQL dialect 1, but as 64-bit integers in SQL dialect 3.</td>
</tr>
<tr>
<td>301</td>
<td>336003084</td>
<td>dsql_warn_precision_ambiguous2</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>335544807</td>
<td>sqlwarn</td>
<td>SQL warning code = @1</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>---------</td>
<td>--------------</td>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>106</td>
<td>336068855</td>
<td>dyn_miss PRIV_WARNING</td>
<td>Warning: @1 on @2 is not granted to @3.</td>
</tr>
<tr>
<td>101</td>
<td>335544366</td>
<td>segment</td>
<td>segment buffer length shorter than expected</td>
</tr>
<tr>
<td>100</td>
<td>335544338</td>
<td>from_no_match</td>
<td>no match for first value expression</td>
</tr>
<tr>
<td>100</td>
<td>335544354</td>
<td>no_record</td>
<td>invalid database key</td>
</tr>
<tr>
<td>100</td>
<td>335544367</td>
<td>segstr_eof</td>
<td>attempted retrieval of more segments than exist</td>
</tr>
<tr>
<td>0</td>
<td>335544875</td>
<td>bad_debug_format</td>
<td>Bad debug info format</td>
</tr>
<tr>
<td>0</td>
<td>335544931</td>
<td>montabexh</td>
<td>Monitoring table space exhausted</td>
</tr>
<tr>
<td>-84</td>
<td>335544554</td>
<td>nonsql_security_rel</td>
<td>object has non-SQL security class defined</td>
</tr>
<tr>
<td>-84</td>
<td>335544555</td>
<td>nonsql_security fld</td>
<td>column has non-SQL security class defined</td>
</tr>
<tr>
<td>-84</td>
<td>335544668</td>
<td>dsql_procedure_use_err</td>
<td>procedure @1 does not return any values</td>
</tr>
<tr>
<td>-85</td>
<td>335544747</td>
<td>username too long</td>
<td>The username entered is too long. Maximum length is 31 bytes.</td>
</tr>
<tr>
<td>-85</td>
<td>335544748</td>
<td>password too long</td>
<td>The password specified is too long. Maximum length is 8 bytes.</td>
</tr>
<tr>
<td>-85</td>
<td>335544749</td>
<td>username_required</td>
<td>A username is required for this operation.</td>
</tr>
<tr>
<td>-85</td>
<td>335544750</td>
<td>password_required</td>
<td>A password is required for this operation</td>
</tr>
<tr>
<td>-85</td>
<td>335544751</td>
<td>bad_protocol</td>
<td>The network protocol specified is invalid</td>
</tr>
<tr>
<td>-85</td>
<td>335544752</td>
<td>dup_username_found</td>
<td>A duplicate user name was found in the security database</td>
</tr>
<tr>
<td>-85</td>
<td>335544753</td>
<td>username_not_found</td>
<td>The user name specified was not found in the security database</td>
</tr>
<tr>
<td>-85</td>
<td>335544754</td>
<td>error_adding_sec_record</td>
<td>An error occurred while attempting to add the user.</td>
</tr>
<tr>
<td>-85</td>
<td>335544755</td>
<td>error_modifying_sec_record</td>
<td>An error occurred while attempting to modify the user record.</td>
</tr>
<tr>
<td>-85</td>
<td>335544756</td>
<td>error_deleting_sec_record</td>
<td>An error occurred while attempting to delete the user record.</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-85</td>
<td>335544757</td>
<td>error_updating_sec_db</td>
<td>An error occurred while updating the security database.</td>
</tr>
<tr>
<td>-103</td>
<td>335544571</td>
<td>dsqI_constant_err</td>
<td>Data type for constant unknown</td>
</tr>
<tr>
<td>-104</td>
<td>335544343</td>
<td>invalid_blr</td>
<td>invalid request BLR at offset @1</td>
</tr>
<tr>
<td>-104</td>
<td>335544390</td>
<td>syntaxerr</td>
<td>BLR syntax error: expected @1 at offset @2, encountered @3</td>
</tr>
<tr>
<td>-104</td>
<td>335544425</td>
<td>ctxinuse</td>
<td>context already in use (BLR error)</td>
</tr>
<tr>
<td>-104</td>
<td>335544426</td>
<td>ctxnotdef</td>
<td>context not defined (BLR error)</td>
</tr>
<tr>
<td>-104</td>
<td>335544429</td>
<td>badparnum</td>
<td>undefined parameter number</td>
</tr>
<tr>
<td>-104</td>
<td>335544440</td>
<td>bad_msg_vec</td>
<td>invalid slice description language at offset @1</td>
</tr>
<tr>
<td>-104</td>
<td>335544457</td>
<td>dsqI_command_err</td>
<td>Invalid command</td>
</tr>
<tr>
<td>-104</td>
<td>335544579</td>
<td>dsqI_internal_err</td>
<td>Internal error</td>
</tr>
<tr>
<td>-104</td>
<td>335544590</td>
<td>dsqI_dup_option</td>
<td>Option specified more than once</td>
</tr>
<tr>
<td>-104</td>
<td>335544591</td>
<td>dsqI_tran_err</td>
<td>Unknown transaction option</td>
</tr>
<tr>
<td>-104</td>
<td>335544592</td>
<td>dsqI_invalid_array</td>
<td>Invalid array reference</td>
</tr>
<tr>
<td>-104</td>
<td>335544608</td>
<td>command_end_err</td>
<td>Unexpected end of command</td>
</tr>
<tr>
<td>-104</td>
<td>335544612</td>
<td>token_err</td>
<td>Token unknown</td>
</tr>
<tr>
<td>-104</td>
<td>335544634</td>
<td>dsqI_token_unk_err</td>
<td>Token unknown - line @1, column @2</td>
</tr>
<tr>
<td>-104</td>
<td>335544709</td>
<td>dsqI_agg_ref_err</td>
<td>Invalid aggregate reference</td>
</tr>
<tr>
<td>-104</td>
<td>335544714</td>
<td>invalid_array_id</td>
<td>invalid blob id</td>
</tr>
<tr>
<td>-104</td>
<td>335544730</td>
<td>cse_not_supported</td>
<td>Client/Server Express not supported in this release</td>
</tr>
<tr>
<td>-104</td>
<td>335544743</td>
<td>token_too_long</td>
<td>token size exceeds limit</td>
</tr>
<tr>
<td>-104</td>
<td>335544763</td>
<td>invalid_string_constant</td>
<td>a string constant is delimited by double quotes</td>
</tr>
<tr>
<td>-104</td>
<td>335544764</td>
<td>transitional_date</td>
<td>DATE must be changed to TIMESTAMP</td>
</tr>
<tr>
<td>-104</td>
<td>335544796</td>
<td>sql_dialect_datatype_unsupport</td>
<td>Client SQL dialect @1 does not support reference to @2 datatype</td>
</tr>
<tr>
<td>-104</td>
<td>335544798</td>
<td>depend_on_uncommitted_rel</td>
<td>You created an indirect dependency on uncommitted metadata. You must roll back the current transaction.</td>
</tr>
<tr>
<td>-104</td>
<td>335544821</td>
<td>dsqI_column_pos_err</td>
<td>Invalid column position used in the @1 clause</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-104</td>
<td>335544822</td>
<td>dsq_agg_where_err</td>
<td>Cannot use an aggregate or window function in a WHERE clause, use HAVING (for aggregate only) instead</td>
</tr>
<tr>
<td>-104</td>
<td>335544823</td>
<td>dsq_agg_group_err</td>
<td>Cannot use an aggregate or window function in a GROUP BY clause</td>
</tr>
<tr>
<td>-104</td>
<td>335544824</td>
<td>dsq_agg_column_err</td>
<td>Invalid expression in the @1 (not contained in either an aggregate function or the GROUP BY clause)</td>
</tr>
<tr>
<td>-104</td>
<td>335544825</td>
<td>dsq_agg_having_err</td>
<td>Invalid expression in the @1 (neither an aggregate function nor a part of the GROUP BY clause)</td>
</tr>
<tr>
<td>-104</td>
<td>335544826</td>
<td>dsq_agg_nested_err</td>
<td>Nested aggregate and window functions are not allowed</td>
</tr>
<tr>
<td>-104</td>
<td>335544849</td>
<td>malformed_string</td>
<td>Malformed string</td>
</tr>
<tr>
<td>-104</td>
<td>335544851</td>
<td>command_end_err2</td>
<td>Unexpected end of command - line @1, column @2</td>
</tr>
<tr>
<td>-104</td>
<td>335544930</td>
<td>too_big_blr</td>
<td>BLR stream length @1 exceeds implementation limit @2</td>
</tr>
<tr>
<td>-104</td>
<td>335544980</td>
<td>internal_rejected_params</td>
<td>Incorrect parameters provided to internal function @1</td>
</tr>
<tr>
<td>-104</td>
<td>335545022</td>
<td>cannot_copy_stmt</td>
<td>Cannot copy statement @1</td>
</tr>
<tr>
<td>-104</td>
<td>335545023</td>
<td>invalid_boolean_usage</td>
<td>Invalid usage of boolean expression</td>
</tr>
<tr>
<td>-104</td>
<td>335545035</td>
<td>svc_no_stdin</td>
<td>No isc_info_svc_stdin in user request, but service thread requested stdin data</td>
</tr>
<tr>
<td>-104</td>
<td>335545037</td>
<td>svc_no_switches</td>
<td>All services except for getting server log require switches</td>
</tr>
<tr>
<td>-104</td>
<td>335545038</td>
<td>svc_bad_size</td>
<td>Size of stdin data is more than was requested from client</td>
</tr>
<tr>
<td>-104</td>
<td>335545039</td>
<td>no_crypt_plugin</td>
<td>Crypt plugin @1 failed to load</td>
</tr>
<tr>
<td>-104</td>
<td>335545040</td>
<td>cp_name_too_long</td>
<td>Length of crypt plugin name should not exceed @1 bytes</td>
</tr>
<tr>
<td>-104</td>
<td>335545045</td>
<td>null_spb</td>
<td>NULL data with non-zero SPB length</td>
</tr>
<tr>
<td>-104</td>
<td>336003075</td>
<td>dsq_transitional_numeric</td>
<td>Precision 10 to 18 changed from DOUBLE PRECISION in SQL dialect 1 to 64-bit scaled integer in SQL dialect 3</td>
</tr>
<tr>
<td>-104</td>
<td>336003077</td>
<td>sql_db_dialect_dtype_unsupport</td>
<td>Database SQL dialect @1 does not support reference to @2 datatype</td>
</tr>
<tr>
<td>-104</td>
<td>336003087</td>
<td>dsq_invalid_label</td>
<td>Label @1 @2 in the current scope</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>-----------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-104</td>
<td>336003088</td>
<td>dsql_datatypes_not_comparable</td>
<td>Datatypes @1 are not comparable in expression @2</td>
</tr>
<tr>
<td>-104</td>
<td>336397215</td>
<td>dsql_max_sort_items</td>
<td>cannot sort on more than 255 items</td>
</tr>
<tr>
<td>-104</td>
<td>336397216</td>
<td>dsql_max_group_items</td>
<td>cannot group on more than 255 items</td>
</tr>
<tr>
<td>-104</td>
<td>336397217</td>
<td>dsql_conflicting_sort_field</td>
<td>Cannot include the same field (@1.@2) twice in the ORDER BY clause with conflicting sorting options</td>
</tr>
<tr>
<td>-104</td>
<td>336397218</td>
<td>dsql_derived_table_more_columns</td>
<td>column list from derived table @1 has more columns than the number of items in its SELECT statement</td>
</tr>
<tr>
<td>-104</td>
<td>336397219</td>
<td>dsql_derived_table_less_columns</td>
<td>column list from derived table @1 has less columns than the number of items in its SELECT statement</td>
</tr>
<tr>
<td>-104</td>
<td>336397220</td>
<td>dsql_derived_field_unnamed</td>
<td>no column name specified for column number @1 in derived table @2</td>
</tr>
<tr>
<td>-104</td>
<td>336397221</td>
<td>dsql_derived_field_dup_name</td>
<td>column @1 was specified multiple times for derived table @2</td>
</tr>
<tr>
<td>-104</td>
<td>336397222</td>
<td>dsql_derived_alias_select</td>
<td>Internal dsql error: alias type expected by pass1_expand_select_node</td>
</tr>
<tr>
<td>-104</td>
<td>336397223</td>
<td>dsql_derived_alias_field</td>
<td>Internal dsql error: alias type expected by pass1_field</td>
</tr>
<tr>
<td>-104</td>
<td>336397224</td>
<td>dsql_auto_field_bad_pos</td>
<td>Internal dsql error: column position out of range in pass1_union_auto_cast</td>
</tr>
<tr>
<td>-104</td>
<td>336397225</td>
<td>dsql_cte_wrong_reference</td>
<td>Recursive CTE member (@1) can refer itself only in FROM clause</td>
</tr>
<tr>
<td>-104</td>
<td>336397226</td>
<td>dsql_cte_cycle</td>
<td>CTE '@1' has cyclic dependencies</td>
</tr>
<tr>
<td>-104</td>
<td>336397227</td>
<td>dsql_cte_outer_join</td>
<td>Recursive member of CTE can't be member of an outer join</td>
</tr>
<tr>
<td>-104</td>
<td>336397228</td>
<td>dsql_cte_mult_references</td>
<td>Recursive member of CTE can't reference itself more than once</td>
</tr>
<tr>
<td>-104</td>
<td>336397229</td>
<td>dsql_cte_not_a_union</td>
<td>Recursive CTE (@1) must be an UNION</td>
</tr>
<tr>
<td>-104</td>
<td>336397230</td>
<td>dsql_cte_nonrecurs_after_recurs</td>
<td>CTE '@1' defined non-recursive member after recursive</td>
</tr>
<tr>
<td>-104</td>
<td>336397231</td>
<td>dsql_cte_wrong_clause</td>
<td>Recursive member of CTE '@1' has @2 clause</td>
</tr>
<tr>
<td>-104</td>
<td>336397232</td>
<td>dsql_cte_union_all</td>
<td>Recursive members of CTE (@1) must be linked with another members via UNION ALL</td>
</tr>
</tbody>
</table>
### Appendix B: Exception Codes and Messages

<table>
<thead>
<tr>
<th>SQL-CODE</th>
<th>GDSCODE</th>
<th>Symbol</th>
<th>Message Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>-104</td>
<td>336397233</td>
<td>dsql_cte_miss_nonrecursive</td>
<td>Non-recursive member is missing in CTE '@1'</td>
</tr>
<tr>
<td>-104</td>
<td>336397234</td>
<td>dsql_cte_nested_with</td>
<td>WITH clause can't be nested</td>
</tr>
<tr>
<td>-104</td>
<td>336397235</td>
<td>dsql_col_more_than_once_using</td>
<td>column @1 appears more than once in USING clause</td>
</tr>
<tr>
<td>-104</td>
<td>336397237</td>
<td>dsql_ce_not_used</td>
<td>CTE &quot;@1&quot; is not used in query</td>
</tr>
<tr>
<td>-104</td>
<td>336397238</td>
<td>dsql_col_more_than_once_view</td>
<td>column @1 appears more than once in ALTER VIEW</td>
</tr>
<tr>
<td>-104</td>
<td>336397257</td>
<td>dsql_max_distinct_items</td>
<td>Cannot have more than 255 items in DISTINCT list</td>
</tr>
<tr>
<td>-104</td>
<td>336397321</td>
<td>dsql_ce_recursive_aggregate</td>
<td>Recursive member of CTE cannot use aggregate or window function</td>
</tr>
<tr>
<td>-104</td>
<td>336397326</td>
<td>dsql_wlock_simple</td>
<td>WITH LOCK can be used only with a single physical table</td>
</tr>
<tr>
<td>-104</td>
<td>336397327</td>
<td>dsql_firstskip_rows</td>
<td>FIRST/SKIP cannot be used with OFFSET/FETCH or ROWS</td>
</tr>
<tr>
<td>-104</td>
<td>336397328</td>
<td>dsql_wlock_aggregates</td>
<td>WITH LOCK cannot be used with aggregates</td>
</tr>
<tr>
<td>-104</td>
<td>336397329</td>
<td>dsql_wlock_conflict</td>
<td>WITH LOCK cannot be used with @1</td>
</tr>
<tr>
<td>-105</td>
<td>335544702</td>
<td>escape_invalid</td>
<td>Invalid ESCAPE sequence</td>
</tr>
<tr>
<td>-105</td>
<td>335544789</td>
<td>extract_input_mismatch</td>
<td>Specified EXTRACT part does not exist in input datatype</td>
</tr>
<tr>
<td>-105</td>
<td>335544884</td>
<td>invalid_similar_pattern</td>
<td>Invalid SIMILAR TO pattern</td>
</tr>
<tr>
<td>-150</td>
<td>335544360</td>
<td>read_only_rel</td>
<td>attempted update of read-only table</td>
</tr>
<tr>
<td>-150</td>
<td>335544362</td>
<td>read_only_view</td>
<td>cannot update read-only view @1</td>
</tr>
<tr>
<td>-150</td>
<td>335544446</td>
<td>non_updatable</td>
<td>not updatable</td>
</tr>
<tr>
<td>-150</td>
<td>335544546</td>
<td>constant_on_view</td>
<td>Cannot define constraints on views</td>
</tr>
<tr>
<td>-151</td>
<td>335544359</td>
<td>read_only_field</td>
<td>attempted update of read-only column</td>
</tr>
<tr>
<td>-155</td>
<td>335544658</td>
<td>dsql_base_table</td>
<td>@1 is not a valid base table of the specified view</td>
</tr>
<tr>
<td>-157</td>
<td>335544598</td>
<td>specify_field_err</td>
<td>must specify column name for view select expression</td>
</tr>
<tr>
<td>-158</td>
<td>335544599</td>
<td>num_field_err</td>
<td>number of columns does not match select list</td>
</tr>
<tr>
<td>-162</td>
<td>335544685</td>
<td>no_dbkey</td>
<td>dbkey not available for multi-table views</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-170</td>
<td>335544512</td>
<td>prcmismat</td>
<td>Input parameter mismatch for procedure @1</td>
</tr>
<tr>
<td>-170</td>
<td>335544619</td>
<td>extern_func_err</td>
<td>External functions cannot have more than 10 parameters</td>
</tr>
<tr>
<td>-170</td>
<td>335544850</td>
<td>prc_out_param_mismatch</td>
<td>Output parameter mismatch for procedure @1</td>
</tr>
<tr>
<td>-170</td>
<td>335545101</td>
<td>fun_param_mismatch</td>
<td>Input parameter mismatch for function @1</td>
</tr>
<tr>
<td>-171</td>
<td>335544439</td>
<td>funmismat</td>
<td>function @1 could not be matched</td>
</tr>
<tr>
<td>-171</td>
<td>335544458</td>
<td>invalid_dimension</td>
<td>column not array or invalid dimensions (expected @1, encountered @2)</td>
</tr>
<tr>
<td>-171</td>
<td>335544618</td>
<td>return_mode_err</td>
<td>Return mode by value not allowed for this data type</td>
</tr>
<tr>
<td>-171</td>
<td>335544873</td>
<td>array_max_dimensions</td>
<td>Array data type can use up to @1 dimensions</td>
</tr>
<tr>
<td>-172</td>
<td>335544438</td>
<td>funnotdef</td>
<td>function @1 is not defined</td>
</tr>
<tr>
<td>-172</td>
<td>335544932</td>
<td>modnotfound</td>
<td>module name or entrypoint could not be found</td>
</tr>
<tr>
<td>-203</td>
<td>335544708</td>
<td>dyn_fld_ambiguous</td>
<td>Ambiguous column reference.</td>
</tr>
<tr>
<td>-204</td>
<td>335544463</td>
<td>gennotdef</td>
<td>generator @1 is not defined</td>
</tr>
<tr>
<td>-204</td>
<td>335544502</td>
<td>stream_not_defined</td>
<td>reference to invalid stream number</td>
</tr>
<tr>
<td>-204</td>
<td>335544509</td>
<td>charset_not_found</td>
<td>CHARACTER SET @1 is not defined</td>
</tr>
<tr>
<td>-204</td>
<td>335544511</td>
<td>prcnotdef</td>
<td>procedure @1 is not defined</td>
</tr>
<tr>
<td>-204</td>
<td>335544515</td>
<td>codnotdef</td>
<td>status code @1 unknown</td>
</tr>
<tr>
<td>-204</td>
<td>335544516</td>
<td>xcpnotdef</td>
<td>exception @1 not defined</td>
</tr>
<tr>
<td>-204</td>
<td>335544532</td>
<td>ref_cnstrnt_notfound</td>
<td>Name of Referential Constraint not defined in constraints table.</td>
</tr>
<tr>
<td>-204</td>
<td>335544551</td>
<td>grant_obj_notfound</td>
<td>could not find object for GRANT</td>
</tr>
<tr>
<td>-204</td>
<td>335544568</td>
<td>text_subtype</td>
<td>Implementation of text subtype @1 not located.</td>
</tr>
<tr>
<td>-204</td>
<td>335544573</td>
<td>dsql_datatype_err</td>
<td>Data type unknown</td>
</tr>
<tr>
<td>-204</td>
<td>335544580</td>
<td>dsql_relation_err</td>
<td>Table unknown</td>
</tr>
<tr>
<td>-204</td>
<td>335544581</td>
<td>dsql_procedure_err</td>
<td>Procedure unknown</td>
</tr>
<tr>
<td>-204</td>
<td>335544588</td>
<td>collation_not_found</td>
<td>COLLATION @1 for CHARACTER SET @2 is not defined</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>---------------------------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>-204</td>
<td>335544589</td>
<td>collation_not_for_charset</td>
<td>COLLATION @1 is not valid for specified CHARACTER SET</td>
</tr>
<tr>
<td>-204</td>
<td>335544595</td>
<td>dsq1_trigger_err</td>
<td>Trigger unknown</td>
</tr>
<tr>
<td>-204</td>
<td>335544620</td>
<td>alias_conflict_err</td>
<td>alias @1 conflicts with an alias in the same statement</td>
</tr>
<tr>
<td>-204</td>
<td>335544621</td>
<td>procedure_conflict_error</td>
<td>alias @1 conflicts with a procedure in the same statement</td>
</tr>
<tr>
<td>-204</td>
<td>335544622</td>
<td>relation_conflict_err</td>
<td>alias @1 conflicts with a table in the same statement</td>
</tr>
<tr>
<td>-204</td>
<td>335544635</td>
<td>dsq1_no_relation_alias</td>
<td>there is no alias or table named @1 at this scope level</td>
</tr>
<tr>
<td>-204</td>
<td>335544636</td>
<td>indexname</td>
<td>there is no index @1 for table @2</td>
</tr>
<tr>
<td>-204</td>
<td>335544640</td>
<td>collation_requires_text</td>
<td>Invalid use of CHARACTER SET or COLLATE</td>
</tr>
<tr>
<td>-204</td>
<td>335544662</td>
<td>dsq1_blob_type_unknown</td>
<td>BLOB SUB_TYPE @1 is not defined</td>
</tr>
<tr>
<td>-204</td>
<td>335544759</td>
<td>bad_default_value</td>
<td>can not define a not null column with NULL as default value</td>
</tr>
<tr>
<td>-204</td>
<td>335544760</td>
<td>invalid_clause</td>
<td>invalid clause --- ’@1’</td>
</tr>
<tr>
<td>-204</td>
<td>335544800</td>
<td>too_many_contexts</td>
<td>Too many Contexts of Relation/Procedure/Views. Maximum allowed is 256</td>
</tr>
<tr>
<td>-204</td>
<td>335544817</td>
<td>bad_limit_param</td>
<td>Invalid parameter to FETCH or FIRST. Only integers &gt;= 0 are allowed.</td>
</tr>
<tr>
<td>-204</td>
<td>335544818</td>
<td>bad_skip_param</td>
<td>Invalid parameter to OFFSET or SKIP. Only integers &gt;= 0 are allowed.</td>
</tr>
<tr>
<td>-204</td>
<td>335544837</td>
<td>bad_substring_offset</td>
<td>Invalid offset parameter @1 to SUBSTRING. Only positive integers are allowed.</td>
</tr>
<tr>
<td>-204</td>
<td>335544853</td>
<td>bad_substring_length</td>
<td>Invalid length parameter @1 to SUBSTRING. Negative integers are not allowed.</td>
</tr>
<tr>
<td>-204</td>
<td>335544854</td>
<td>charset_not_installed</td>
<td>CHARACTER SET @1 is not installed</td>
</tr>
<tr>
<td>-204</td>
<td>335544855</td>
<td>collation_not_installed</td>
<td>COLLATION @1 for CHARACTER SET @2 is not installed</td>
</tr>
<tr>
<td>-204</td>
<td>335544867</td>
<td>subtype_for_internal_use</td>
<td>Blob sub_types bigger than 1 (text) are for internal use only</td>
</tr>
<tr>
<td>-204</td>
<td>335545104</td>
<td>invalid_attachment_charset</td>
<td>CHARACTER SET @1 cannot be used as a attachment character set</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-204</td>
<td>336003085</td>
<td>dsq_ambiguous_field_name</td>
<td>Ambiguous field name between @1 and @2</td>
</tr>
<tr>
<td>-205</td>
<td>335544396</td>
<td>fldnotdef</td>
<td>column @1 is not defined in table @2</td>
</tr>
<tr>
<td>-205</td>
<td>335544552</td>
<td>grantFldNotfound</td>
<td>could not find column for GRANT</td>
</tr>
<tr>
<td>-205</td>
<td>335544883</td>
<td>fldnotdef2</td>
<td>column @1 is not defined in procedure @2</td>
</tr>
<tr>
<td>-206</td>
<td>335544578</td>
<td>dsq_field_err</td>
<td>Column unknown</td>
</tr>
<tr>
<td>-206</td>
<td>335544587</td>
<td>dsq_blob_err</td>
<td>Column is not a BLOB</td>
</tr>
<tr>
<td>-206</td>
<td>335544596</td>
<td>dsq_subselect_err</td>
<td>Subselect illegal in this context</td>
</tr>
<tr>
<td>-206</td>
<td>336397208</td>
<td>dsq_line_col_error</td>
<td>At line @1, column @2</td>
</tr>
<tr>
<td>-206</td>
<td>336397209</td>
<td>dsq_unknown_pos</td>
<td>At unknown line and column</td>
</tr>
<tr>
<td>-206</td>
<td>336397210</td>
<td>dsq_no_dup_name</td>
<td>Column @1 cannot be repeated in @2 statement</td>
</tr>
<tr>
<td>-208</td>
<td>335544617</td>
<td>order_by_err</td>
<td>invalid ORDER BY clause</td>
</tr>
<tr>
<td>-219</td>
<td>335544395</td>
<td>relnotdef</td>
<td>table @1 is not defined</td>
</tr>
<tr>
<td>-219</td>
<td>335544872</td>
<td>domnotdef</td>
<td>domain @1 is not defined</td>
</tr>
<tr>
<td>-230</td>
<td>335544487</td>
<td>walw_err</td>
<td>WAL Writer error</td>
</tr>
<tr>
<td>-231</td>
<td>335544488</td>
<td>logh_small</td>
<td>Log file header of @1 too small</td>
</tr>
<tr>
<td>-232</td>
<td>335544489</td>
<td>logh_inv_version</td>
<td>Invalid version of log file @1</td>
</tr>
<tr>
<td>-233</td>
<td>335544490</td>
<td>logh_open_flag</td>
<td>Log file @1 not latest in the chain but open flag still set</td>
</tr>
<tr>
<td>-234</td>
<td>335544491</td>
<td>logh_open_flag2</td>
<td>Log file @1 not closed properly; database recovery may be required</td>
</tr>
<tr>
<td>-235</td>
<td>335544492</td>
<td>logh_diff_dbname</td>
<td>Database name in the log file @1 is different</td>
</tr>
<tr>
<td>-236</td>
<td>335544493</td>
<td>logf_unexpected_eof</td>
<td>Unexpected end of log file @1 at offset @2</td>
</tr>
<tr>
<td>-237</td>
<td>335544494</td>
<td>logr_incomplete</td>
<td>Incomplete log record at offset @1 in log file @2</td>
</tr>
<tr>
<td>-238</td>
<td>335544495</td>
<td>logr_header_small</td>
<td>Log record header too small at offset @1 in log file @2</td>
</tr>
<tr>
<td>-239</td>
<td>335544496</td>
<td>logb_small</td>
<td>Log block too small at offset @1 in log file @2</td>
</tr>
<tr>
<td>-239</td>
<td>335544691</td>
<td>cache_too_small</td>
<td>Insufficient memory to allocate page buffer cache</td>
</tr>
<tr>
<td>-239</td>
<td>335544693</td>
<td>log_too_small</td>
<td>Log size too small</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-239</td>
<td>335544694</td>
<td>partition_too_small</td>
<td>Log partition size too small</td>
</tr>
<tr>
<td>-240</td>
<td>335544497</td>
<td>wal_illegal_attach</td>
<td>Illegal attempt to attach to an uninitialized WAL segment for @1</td>
</tr>
<tr>
<td>-241</td>
<td>335544498</td>
<td>wal_invalid_wpb</td>
<td>Invalid WAL parameter block option @1</td>
</tr>
<tr>
<td>-242</td>
<td>335544499</td>
<td>wal_err_rollover</td>
<td>Cannot roll over to the next log file @1</td>
</tr>
<tr>
<td>-243</td>
<td>335544500</td>
<td>no_wal</td>
<td>database does not use Write-ahead Log</td>
</tr>
<tr>
<td>-244</td>
<td>335544503</td>
<td>wal_subsys_error</td>
<td>WAL subsystem encountered error</td>
</tr>
<tr>
<td>-245</td>
<td>335544504</td>
<td>wal_subsys_corrupt</td>
<td>WAL subsystem corrupted</td>
</tr>
<tr>
<td>-246</td>
<td>335544513</td>
<td>wal_bugcheck</td>
<td>Database @1: WAL subsystem bug for pid @2 @3</td>
</tr>
<tr>
<td>-247</td>
<td>335544514</td>
<td>wal_cant_expand</td>
<td>Could not expand the WAL segment for database @1</td>
</tr>
<tr>
<td>-248</td>
<td>335544521</td>
<td>wal_err_rollover2</td>
<td>Unable to roll over please see Firebird log.</td>
</tr>
<tr>
<td>-249</td>
<td>335544522</td>
<td>wal_err_logwrite</td>
<td>WAL I/O error. Please see Firebird log.</td>
</tr>
<tr>
<td>-250</td>
<td>335544523</td>
<td>wal_err_jrn_comm</td>
<td>WAL writer - Journal server communication error. Please see Firebird log.</td>
</tr>
<tr>
<td>-251</td>
<td>335544524</td>
<td>wal_err_expansion</td>
<td>WAL buffers cannot be increased. Please see Firebird log.</td>
</tr>
<tr>
<td>-252</td>
<td>335544525</td>
<td>wal_err_setup</td>
<td>WAL setup error. Please see Firebird log.</td>
</tr>
<tr>
<td>-253</td>
<td>335544526</td>
<td>wal_err_ww_sync</td>
<td>obsolete</td>
</tr>
<tr>
<td>-254</td>
<td>335544527</td>
<td>wal_err_ww_start</td>
<td>Cannot start WAL writer for the database @1</td>
</tr>
<tr>
<td>-255</td>
<td>335544556</td>
<td>wal_cache_err</td>
<td>Write-ahead Log without shared cache configuration not allowed</td>
</tr>
<tr>
<td>-257</td>
<td>335544566</td>
<td>start_cm_for_wal</td>
<td>WAL defined; Cache Manager must be started first</td>
</tr>
<tr>
<td>-258</td>
<td>335544567</td>
<td>wal_ovflow_log_required</td>
<td>Overflow log specification required for round-robin log</td>
</tr>
<tr>
<td>-259</td>
<td>335544629</td>
<td>wal_shadow_err</td>
<td>Write-ahead Log with shadowing configuration not allowed</td>
</tr>
<tr>
<td>-260</td>
<td>335544690</td>
<td>cache_redef</td>
<td>Cache redefined</td>
</tr>
<tr>
<td>-260</td>
<td>335544692</td>
<td>log_redef</td>
<td>Log redefined</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-261</td>
<td>335544695</td>
<td>partition_not_supp</td>
<td>Partitions not supported in series of log file specification</td>
</tr>
<tr>
<td>-261</td>
<td>335544696</td>
<td>log_length_spec</td>
<td>Total length of a partitioned log must be specified</td>
</tr>
<tr>
<td>-281</td>
<td>335544637</td>
<td>no_stream_plan</td>
<td>table @1 is not referenced in plan</td>
</tr>
<tr>
<td>-282</td>
<td>335544638</td>
<td>stream_twice</td>
<td>table @1 is referenced more than once in plan; use aliases to distinguish</td>
</tr>
<tr>
<td>-282</td>
<td>335544643</td>
<td>dsq_self_join</td>
<td>the table @1 is referenced twice; use aliases to differentiate</td>
</tr>
<tr>
<td>-282</td>
<td>335544659</td>
<td>duplicate_base_table</td>
<td>table @1 is referenced twice in view; use an alias to distinguish</td>
</tr>
<tr>
<td>-282</td>
<td>335544660</td>
<td>view_alias</td>
<td>view @1 has more than one base table; use aliases to distinguish</td>
</tr>
<tr>
<td>-282</td>
<td>335544710</td>
<td>complex_view</td>
<td>navigational stream @1 references a view with more than one base table</td>
</tr>
<tr>
<td>-283</td>
<td>335544639</td>
<td>stream_not_found</td>
<td>table @1 is referenced in the plan but not the from list</td>
</tr>
<tr>
<td>-284</td>
<td>335544642</td>
<td>index_unused</td>
<td>index @1 cannot be used in the specified plan</td>
</tr>
<tr>
<td>-291</td>
<td>335544531</td>
<td>primary_key_notnull</td>
<td>Column used in a PRIMARY constraint must be NOT NULL.</td>
</tr>
<tr>
<td>-291</td>
<td>335545103</td>
<td>domain_primary_key_notnull</td>
<td>Domain used in the PRIMARY KEY constraint of table @1 must be NOT NULL</td>
</tr>
<tr>
<td>-292</td>
<td>335544534</td>
<td>ref_cnstrnt_update</td>
<td>Cannot update constraints (RDB$REF_CONSTRAINTS).</td>
</tr>
<tr>
<td>-293</td>
<td>335544535</td>
<td>check_cnstrnt_update</td>
<td>Cannot update constraints (RDB$CHECK_CONSTRAINTS).</td>
</tr>
<tr>
<td>-294</td>
<td>335544536</td>
<td>check_cnstrnt_del</td>
<td>Cannot delete CHECK constraint entry (RDB$CHECK_CONSTRAINTS)</td>
</tr>
<tr>
<td>-295</td>
<td>335544545</td>
<td>rel_cnstrnt_update</td>
<td>Cannot update constraints (RDB$RELATION_CONSTRAINTS).</td>
</tr>
<tr>
<td>-296</td>
<td>335544547</td>
<td>invld_cnstrnt_type</td>
<td>internal Firebird consistency check (invalid RDB$CONSTRAINT_TYPE)</td>
</tr>
<tr>
<td>-297</td>
<td>335544558</td>
<td>check_constraint</td>
<td>Operation violates CHECK constraint @1 on view or table @2</td>
</tr>
<tr>
<td>-313</td>
<td>335544669</td>
<td>dsq_count_mismatch</td>
<td>count of column list and variable list do not match</td>
</tr>
<tr>
<td>SQL CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-313</td>
<td>336003099</td>
<td>upd_ins_doesnt_match_pk</td>
<td>UPDATE OR INSERT field list does not match primary key of table @1</td>
</tr>
<tr>
<td>-313</td>
<td>336003100</td>
<td>upd_ins_doesnt_match_matching</td>
<td>UPDATE OR INSERT field list does not match MATCHING clause</td>
</tr>
<tr>
<td>-313</td>
<td>336003111</td>
<td>dsq1_wrong_param_num</td>
<td>Wrong number of parameters (expected @1, got @2)</td>
</tr>
<tr>
<td>-314</td>
<td>335544565</td>
<td>transliteration_failed</td>
<td>Cannot transliterate character between character sets</td>
</tr>
<tr>
<td>-315</td>
<td>336068815</td>
<td>dyn_dtype_invalid</td>
<td>Cannot change datatype for column @1. Changing datatype is not supported for BLOB or ARRAY columns.</td>
</tr>
<tr>
<td>-383</td>
<td>336068814</td>
<td>dyn_dependency_exists</td>
<td>Column @1 from table @2 is referenced in @3</td>
</tr>
<tr>
<td>-401</td>
<td>335544647</td>
<td>invalid_operator</td>
<td>invalid comparison operator for find operation</td>
</tr>
<tr>
<td>-402</td>
<td>335544368</td>
<td>segstr_no_op</td>
<td>attempted invalid operation on a BLOB</td>
</tr>
<tr>
<td>-402</td>
<td>335544414</td>
<td>blobnotsup</td>
<td>BLOB and array data types are not supported for @1 operation</td>
</tr>
<tr>
<td>-402</td>
<td>335544427</td>
<td>datnotsup</td>
<td>data operation not supported</td>
</tr>
<tr>
<td>-406</td>
<td>335544457</td>
<td>out_of_bounds</td>
<td>subscript out of bounds</td>
</tr>
<tr>
<td>-406</td>
<td>335545028</td>
<td>ss_out_of_bounds</td>
<td>Subscript @1 out of bounds [@2, @3]</td>
</tr>
<tr>
<td>-407</td>
<td>33554435</td>
<td>nullsegkey</td>
<td>null segment of UNIQUE KEY</td>
</tr>
<tr>
<td>-413</td>
<td>335544334</td>
<td>convert_error</td>
<td>conversion error from string &quot;@1&quot;</td>
</tr>
<tr>
<td>-413</td>
<td>335544454</td>
<td>nofilter</td>
<td>filter not found to convert type @1 to type @2</td>
</tr>
<tr>
<td>-413</td>
<td>335544860</td>
<td>blob_convert_error</td>
<td>Unsupported conversion to target type BLOB (subtype @1)</td>
</tr>
<tr>
<td>-413</td>
<td>335544861</td>
<td>array_convert_error</td>
<td>Unsupported conversion to target type ARRAY</td>
</tr>
<tr>
<td>-501</td>
<td>33554577</td>
<td>dsq1_cursor_close_err</td>
<td>Attempt to reclose a closed cursor</td>
</tr>
<tr>
<td>-502</td>
<td>33554574</td>
<td>dsq1_decl_err</td>
<td>Invalid cursor declaration</td>
</tr>
<tr>
<td>-502</td>
<td>33554576</td>
<td>dsq1_cursor_open_err</td>
<td>Attempt to reopen an open cursor</td>
</tr>
<tr>
<td>-502</td>
<td>336003090</td>
<td>dsq1_cursor_redefined</td>
<td>Statement already has a cursor @1 assigned</td>
</tr>
<tr>
<td>-502</td>
<td>336003091</td>
<td>dsq1_cursor_not_found</td>
<td>Cursor @1 is not found in the current context</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>--------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-502</td>
<td>336003092</td>
<td>dsql_cursor_exists</td>
<td>Cursor @1 already exists in the current context</td>
</tr>
<tr>
<td>-502</td>
<td>336003093</td>
<td>dsql_cursor_rel_ambiguous</td>
<td>Relation @1 is ambiguous in cursor @2</td>
</tr>
<tr>
<td>-502</td>
<td>336003094</td>
<td>dsql_cursor_rel_not_found</td>
<td>Relation @1 is not found in cursor @2</td>
</tr>
<tr>
<td>-502</td>
<td>336003095</td>
<td>dsql_cursor_not_open</td>
<td>Cursor is not open</td>
</tr>
<tr>
<td>-504</td>
<td>335544572</td>
<td>dsql_cursor_err</td>
<td>Invalid cursor reference</td>
</tr>
<tr>
<td>-504</td>
<td>336003089</td>
<td>dsql_cursor_invalid</td>
<td>Empty cursor name is not allowed</td>
</tr>
<tr>
<td>-508</td>
<td>335544348</td>
<td>no_cur_rec</td>
<td>no current record for fetch operation</td>
</tr>
<tr>
<td>-510</td>
<td>335544575</td>
<td>dsql_cursor_update_err</td>
<td>Cursor @1 is not updatable</td>
</tr>
<tr>
<td>-518</td>
<td>335544582</td>
<td>dsql_request_err</td>
<td>Request unknown</td>
</tr>
<tr>
<td>-519</td>
<td>335544688</td>
<td>dsql_open_cursor_request</td>
<td>The prepare statement identifies a prepare statement with an open cursor</td>
</tr>
<tr>
<td>-530</td>
<td>335544466</td>
<td>foreign_key</td>
<td>violation of FOREIGN KEY constraint &quot;@1&quot; on table &quot;@2&quot;</td>
</tr>
<tr>
<td>-530</td>
<td>335544838</td>
<td>foreign_key_target_doesnt_exist</td>
<td>Foreign key reference target does not exist</td>
</tr>
<tr>
<td>-530</td>
<td>335544839</td>
<td>foreign_key_references_present</td>
<td>Foreign key references are present for the record</td>
</tr>
<tr>
<td>-531</td>
<td>335544597</td>
<td>dsql_crdb_prepare_err</td>
<td>Cannot prepare a CREATE DATABASE/SCHEMA statement</td>
</tr>
<tr>
<td>-532</td>
<td>335544469</td>
<td>trans_invalid</td>
<td>transaction marked invalid and cannot be committed</td>
</tr>
<tr>
<td>-532</td>
<td>335545002</td>
<td>attachment_in_use</td>
<td>Attachment is in use</td>
</tr>
<tr>
<td>-532</td>
<td>335545003</td>
<td>transaction_in_use</td>
<td>Transaction is in use</td>
</tr>
<tr>
<td>-532</td>
<td>335545017</td>
<td>async_active</td>
<td>Asynchronous call is already running for this attachment</td>
</tr>
<tr>
<td>-551</td>
<td>335544352</td>
<td>no_priv</td>
<td>no permission for @1 access to @2 @3</td>
</tr>
<tr>
<td>-551</td>
<td>335544790</td>
<td>insufficient_svc_privileges</td>
<td>Service @1 requires SYSDBA permissions. Reattach to the Service Manager using the SYSDBA account.</td>
</tr>
<tr>
<td>-551</td>
<td>335545033</td>
<td>trunc_limits</td>
<td>expected length @1, actual @2</td>
</tr>
<tr>
<td>-551</td>
<td>335545034</td>
<td>info_access</td>
<td>Wrong info requested in isc_svc_query() for anonymous service</td>
</tr>
<tr>
<td>-551</td>
<td>335545036</td>
<td>svc_start_failed</td>
<td>Start request for anonymous service is impossible</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-552</td>
<td>335544550</td>
<td>not_rel_owner</td>
<td>only the owner of a table may reassign ownership</td>
</tr>
<tr>
<td>-552</td>
<td>335544553</td>
<td>grant_nopriv</td>
<td>user does not have GRANT privileges for operation</td>
</tr>
<tr>
<td>-552</td>
<td>335544707</td>
<td>grant_nopriv_on_base</td>
<td>user does not have GRANT privileges on base table/view for operation</td>
</tr>
<tr>
<td>-552</td>
<td>335545058</td>
<td>protect_ownership</td>
<td>Only the owner can change the ownership</td>
</tr>
<tr>
<td>-553</td>
<td>335544529</td>
<td>existing_priv_mod</td>
<td>cannot modify an existing user privilege</td>
</tr>
<tr>
<td>-595</td>
<td>335544645</td>
<td>stream_crack</td>
<td>the current position is on a crack</td>
</tr>
<tr>
<td>-596</td>
<td>335544374</td>
<td>stream_eof</td>
<td>attempt to fetch past the last record in a record stream</td>
</tr>
<tr>
<td>-596</td>
<td>335544644</td>
<td>stream_bof</td>
<td>attempt to fetch before the first record in a record stream</td>
</tr>
<tr>
<td>-596</td>
<td>335545092</td>
<td>cursor_not_positioned</td>
<td>Cursor @1 is not positioned in a valid record</td>
</tr>
<tr>
<td>-597</td>
<td>335544632</td>
<td>dsqcl_file_length_err</td>
<td>Preceding file did not specify length, so @1 must include starting page number</td>
</tr>
<tr>
<td>-598</td>
<td>335544633</td>
<td>dsqcl_shadow_number_err</td>
<td>Shadow number must be a positive integer</td>
</tr>
<tr>
<td>-599</td>
<td>335544607</td>
<td>node_err</td>
<td>gen.c: node not supported</td>
</tr>
<tr>
<td>-599</td>
<td>335544625</td>
<td>node_name_err</td>
<td>A node name is not permitted in a secondary, shadow, cache or log file name</td>
</tr>
<tr>
<td>-600</td>
<td>335544680</td>
<td>crrp_data_err</td>
<td>sort error: corruption in data structure</td>
</tr>
<tr>
<td>-601</td>
<td>335544646</td>
<td>db_or_file_exists</td>
<td>database or file exists</td>
</tr>
<tr>
<td>-604</td>
<td>335544593</td>
<td>dsqcl_max_arr_dim_exceeded</td>
<td>Array declared with too many dimensions</td>
</tr>
<tr>
<td>-604</td>
<td>335544594</td>
<td>dsqcl_arr_range_error</td>
<td>Illegal array dimension range</td>
</tr>
<tr>
<td>-605</td>
<td>335544682</td>
<td>dsqcl_field_ref</td>
<td>Inappropriate self-reference of column</td>
</tr>
<tr>
<td>-607</td>
<td>335544351</td>
<td>no_meta_update</td>
<td>unsuccessful metadata update</td>
</tr>
<tr>
<td>-607</td>
<td>335544549</td>
<td>systrig_update</td>
<td>cannot modify or erase a system trigger</td>
</tr>
<tr>
<td>-607</td>
<td>335544657</td>
<td>dsqcl_no_blob_array</td>
<td>Array/BLOB/DATE data types not allowed in arithmetic</td>
</tr>
<tr>
<td>-607</td>
<td>335544746</td>
<td>reftable_requires_pk</td>
<td>&quot;REFERENCES table&quot; without &quot;,(column)&quot; requires PRIMARY KEY on referenced table</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>-----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>-607</td>
<td>335544815</td>
<td>generator_name</td>
<td>GENERATOR @1</td>
</tr>
<tr>
<td>-607</td>
<td>335544816</td>
<td>udf_name</td>
<td>Function @1</td>
</tr>
<tr>
<td>-607</td>
<td>335544858</td>
<td>must_have_phys_field</td>
<td>Can’t have relation with only computed fields or constraints</td>
</tr>
<tr>
<td>-607</td>
<td>336003074</td>
<td>dsql_dbkey_from_non_table</td>
<td>Cannot SELECT RDB$DB_KEY from a stored procedure.</td>
</tr>
<tr>
<td>-607</td>
<td>336003086</td>
<td>dsql_udf_return_pos_err</td>
<td>External function should have return position between 1 and @1</td>
</tr>
<tr>
<td>-607</td>
<td>336003096</td>
<td>dsql_type_not_supp_ext_tab</td>
<td>Data type @1 is not supported for EXTERNAL TABLES. Relation '@2', field '@3'</td>
</tr>
<tr>
<td>-607</td>
<td>336003104</td>
<td>dsql_record_version_table</td>
<td>To be used with RDB$RECORD_VERSION, @1 must be a table or a view of single table</td>
</tr>
<tr>
<td>-607</td>
<td>336068845</td>
<td>dyn_cannot_del_syscoll</td>
<td>Cannot delete system collation</td>
</tr>
<tr>
<td>-607</td>
<td>336068866</td>
<td>dyn_cannot_mod_sysproc</td>
<td>Cannot ALTER or DROP system procedure @1</td>
</tr>
<tr>
<td>-607</td>
<td>336068867</td>
<td>dyn_cannot_mod_systrig</td>
<td>Cannot ALTER or DROP system trigger @1</td>
</tr>
<tr>
<td>-607</td>
<td>336068868</td>
<td>dyn_cannot_mod_sysfunc</td>
<td>Cannot ALTER or DROP system function @1</td>
</tr>
<tr>
<td>-607</td>
<td>336068869</td>
<td>dyn_invalid_ddl_proc</td>
<td>Invalid DDL statement for procedure @1</td>
</tr>
<tr>
<td>-607</td>
<td>336068870</td>
<td>dyn_invalid_ddl_trig</td>
<td>Invalid DDL statement for trigger @1</td>
</tr>
<tr>
<td>-607</td>
<td>336068878</td>
<td>dyn_invalid_ddl_func</td>
<td>Invalid DDL statement for function @1</td>
</tr>
<tr>
<td>-607</td>
<td>336397206</td>
<td>dsql_table_not_found</td>
<td>Table @1 does not exist</td>
</tr>
<tr>
<td>-607</td>
<td>336397207</td>
<td>dsql_view_not_found</td>
<td>View @1 does not exist</td>
</tr>
<tr>
<td>-607</td>
<td>336397212</td>
<td>dsql_no_array_computed</td>
<td>Array and BLOB data types not allowed in computed field</td>
</tr>
<tr>
<td>-607</td>
<td>336397214</td>
<td>dsql_only_can_subscript_array</td>
<td>scalar operator used on field @1 which is not an array</td>
</tr>
<tr>
<td>-612</td>
<td>336068812</td>
<td>dyn_domain_name_exists</td>
<td>Cannot rename domain @1 to @2. A domain with that name already exists.</td>
</tr>
<tr>
<td>-612</td>
<td>336068813</td>
<td>dyn_field_name_exists</td>
<td>Cannot rename column @1 to @2. A column with that name already exists in table @3.</td>
</tr>
<tr>
<td>SQL- CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>-615</td>
<td>335544475</td>
<td>relation_lock</td>
<td>lock on table @1 conflicts with existing lock</td>
</tr>
<tr>
<td>-615</td>
<td>335544476</td>
<td>record_lock</td>
<td>requested record lock conflicts with existing lock</td>
</tr>
<tr>
<td>-615</td>
<td>335544501</td>
<td>drop_wal</td>
<td>cannot drop log file when journaling is enabled</td>
</tr>
<tr>
<td>-615</td>
<td>335544507</td>
<td>range_in_use</td>
<td>refresh range number @1 already in use</td>
</tr>
<tr>
<td>-616</td>
<td>335544530</td>
<td>primary_key_ref</td>
<td>Cannot delete PRIMARY KEY being used in FOREIGN KEY definition.</td>
</tr>
<tr>
<td>-616</td>
<td>335544539</td>
<td>integ_index_del</td>
<td>Cannot delete index used by an Integrity Constraint</td>
</tr>
<tr>
<td>-616</td>
<td>335544540</td>
<td>integ_index_mod</td>
<td>Cannot modify index used by an Integrity Constraint</td>
</tr>
<tr>
<td>-616</td>
<td>335544541</td>
<td>check_trig_del</td>
<td>Cannot delete trigger used by a CHECK Constraint</td>
</tr>
<tr>
<td>-616</td>
<td>335544543</td>
<td>cnstrnt_fld_del</td>
<td>Cannot delete column being used in an Integrity Constraint</td>
</tr>
<tr>
<td>-616</td>
<td>335544630</td>
<td>dependency</td>
<td>there are @1 dependencies</td>
</tr>
<tr>
<td>-616</td>
<td>335544674</td>
<td>del_last_field</td>
<td>last column in a table cannot be deleted</td>
</tr>
<tr>
<td>-616</td>
<td>335544728</td>
<td>integ_index_deactivate</td>
<td>Cannot deactivate index used by an integrity constraint</td>
</tr>
<tr>
<td>-616</td>
<td>335544729</td>
<td>integ_deactivate_primary</td>
<td>Cannot deactivate index used by a PRIMARY/UNIQUE constraint</td>
</tr>
<tr>
<td>-617</td>
<td>335544542</td>
<td>check_trig_update</td>
<td>Cannot update trigger used by a CHECK Constraint</td>
</tr>
<tr>
<td>-617</td>
<td>335544544</td>
<td>cnstrnt_fld_rename</td>
<td>Cannot rename column being used in an Integrity Constraint</td>
</tr>
<tr>
<td>-618</td>
<td>335544537</td>
<td>integ_index_seg_del</td>
<td>Cannot delete index segment used by an Integrity Constraint</td>
</tr>
<tr>
<td>-618</td>
<td>335544538</td>
<td>integ_index_seg_mod</td>
<td>Cannot update index segment used by an Integrity Constraint</td>
</tr>
<tr>
<td>-625</td>
<td>335544347</td>
<td>not_valid</td>
<td>validation error for column @1, value &quot;@2&quot;</td>
</tr>
<tr>
<td>-625</td>
<td>335544879</td>
<td>not_valid_for_var</td>
<td>validation error for variable @1, value &quot;@2&quot;</td>
</tr>
<tr>
<td>-625</td>
<td>335544880</td>
<td>not_valid_for</td>
<td>validation error for @1, value &quot;@2&quot;</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-637</td>
<td>335544664</td>
<td>dsqlduplicate_spec</td>
<td>duplicate specification of @1 - not supported</td>
</tr>
<tr>
<td>-637</td>
<td>336397213</td>
<td>dsqlimplicit_domain_name</td>
<td>Implicit domain name @1 not allowed in user created domain</td>
</tr>
<tr>
<td>-660</td>
<td>335544533</td>
<td>foreign_key_notfound</td>
<td>Non-existent PRIMARY or UNIQUE KEY specified for FOREIGN KEY.</td>
</tr>
<tr>
<td>-660</td>
<td>335544628</td>
<td>idx_create_err</td>
<td>cannot create index @1</td>
</tr>
<tr>
<td>-660</td>
<td>336003098</td>
<td>primary_key_required</td>
<td>Primary key required on table @1</td>
</tr>
<tr>
<td>-663</td>
<td>335544624</td>
<td>idx_seg_err</td>
<td>segment count of 0 defined for index @1</td>
</tr>
<tr>
<td>-663</td>
<td>335544631</td>
<td>idx_key_err</td>
<td>too many keys defined for index @1</td>
</tr>
<tr>
<td>-663</td>
<td>335544672</td>
<td>key_field_err</td>
<td>too few key columns found for index @1 (incorrect column name?)</td>
</tr>
<tr>
<td>-664</td>
<td>335544434</td>
<td>keytoobig</td>
<td>key size exceeds implementation restriction for index &quot;@1&quot;</td>
</tr>
<tr>
<td>-677</td>
<td>335544445</td>
<td>ext_err</td>
<td>@1 extension error</td>
</tr>
<tr>
<td>-685</td>
<td>335544465</td>
<td>bad_segstr_type</td>
<td>invalid BLOB type for operation</td>
</tr>
<tr>
<td>-685</td>
<td>335544670</td>
<td>blob_idx_err</td>
<td>attempt to index BLOB column in index @1</td>
</tr>
<tr>
<td>-685</td>
<td>335544671</td>
<td>array_idx_err</td>
<td>attempt to index array column in index @1</td>
</tr>
<tr>
<td>-689</td>
<td>335544403</td>
<td>badpagtyp</td>
<td>page @1 is of wrong type (expected @2, found @3)</td>
</tr>
<tr>
<td>-689</td>
<td>335544650</td>
<td>page_type_err</td>
<td>wrong page type</td>
</tr>
<tr>
<td>-690</td>
<td>335544679</td>
<td>no_segments_err</td>
<td>segments not allowed in expression index @1</td>
</tr>
<tr>
<td>-691</td>
<td>335544681</td>
<td>rec_size_err</td>
<td>new record size of @1 bytes is too big</td>
</tr>
<tr>
<td>-692</td>
<td>335544477</td>
<td>max_idx</td>
<td>maximum indexes per table (@1) exceeded</td>
</tr>
<tr>
<td>-693</td>
<td>335544663</td>
<td>req_max_clones_exceeded</td>
<td>Too many concurrent executions of the same request</td>
</tr>
<tr>
<td>-694</td>
<td>335544684</td>
<td>no_field_access</td>
<td>cannot access column @1 in view @2</td>
</tr>
<tr>
<td>-802</td>
<td>335544321</td>
<td>arith_except</td>
<td>arithmetic exception, numeric overflow, or string truncation</td>
</tr>
<tr>
<td>-802</td>
<td>335544836</td>
<td>concat_overflow</td>
<td>Concatenation overflow. Resulting string cannot exceed 32765 bytes in length.</td>
</tr>
<tr>
<td>SQLCODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>---------------------------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td>-802</td>
<td>335544914</td>
<td>string_truncation</td>
<td>string right truncation</td>
</tr>
<tr>
<td>-802</td>
<td>335544915</td>
<td>blob_truncation</td>
<td>blob truncation when converting to a string: length limit exceeded</td>
</tr>
<tr>
<td>-802</td>
<td>335544916</td>
<td>numeric_out_of_range</td>
<td>numeric value is out of range</td>
</tr>
<tr>
<td>-802</td>
<td>336003105</td>
<td>dsql_invalid_sqlda_version</td>
<td>SQLDA version expected between @1 and @2, found @3</td>
</tr>
<tr>
<td>-802</td>
<td>336003106</td>
<td>dsql_sqlvar_index</td>
<td>at SQLVAR index @1</td>
</tr>
<tr>
<td>-802</td>
<td>336003107</td>
<td>dsql_no_sqind</td>
<td>empty pointer to NULL indicator variable</td>
</tr>
<tr>
<td>-802</td>
<td>336003108</td>
<td>dsql_no_sqldata</td>
<td>empty pointer to data</td>
</tr>
<tr>
<td>-802</td>
<td>336003109</td>
<td>dsql_no_input_sqlda</td>
<td>No SQLDA for input values provided</td>
</tr>
<tr>
<td>-802</td>
<td>336003110</td>
<td>dsql_no_output_sqlda</td>
<td>No SQLDA for output values provided</td>
</tr>
<tr>
<td>-803</td>
<td>335544349</td>
<td>no_dup</td>
<td>attempt to store duplicate value (visible to active transactions) in unique index &quot;@1&quot;</td>
</tr>
<tr>
<td>-803</td>
<td>335544665</td>
<td>unique_key_violation</td>
<td>violation of PRIMARY or UNIQUE KEY constraint &quot;@1&quot; on table &quot;@2&quot;</td>
</tr>
<tr>
<td>-804</td>
<td>335544380</td>
<td>wronumarg</td>
<td>wrong number of arguments on call</td>
</tr>
<tr>
<td>-804</td>
<td>335544583</td>
<td>dsql_sqlda_err</td>
<td>SQLDA error</td>
</tr>
<tr>
<td>-804</td>
<td>335544584</td>
<td>dsql_var_count_err</td>
<td>Count of read-write columns does not equal count of values</td>
</tr>
<tr>
<td>-804</td>
<td>335544586</td>
<td>dsql_function_err</td>
<td>Function unknown</td>
</tr>
<tr>
<td>-804</td>
<td>335544713</td>
<td>dsql_sqlda_value_err</td>
<td>Incorrect values within SQLDA structure</td>
</tr>
<tr>
<td>-804</td>
<td>335545050</td>
<td>wrong_message_length</td>
<td>Message length passed from user application does not match set of columns</td>
</tr>
<tr>
<td>-804</td>
<td>335545051</td>
<td>no_output_format</td>
<td>Resultset is missing output format information</td>
</tr>
<tr>
<td>-804</td>
<td>335545052</td>
<td>item_finish</td>
<td>Message metadata not ready - item @1 is not finished</td>
</tr>
<tr>
<td>-804</td>
<td>335545100</td>
<td>interface_version_too_old</td>
<td>Interface @3 version too old: expected @1, found @2</td>
</tr>
<tr>
<td>-804</td>
<td>336003097</td>
<td>dsql_feature_not_supported_ods</td>
<td>Feature not supported on ODS version older than @1.@2</td>
</tr>
<tr>
<td>-804</td>
<td>336397205</td>
<td>dsql_too_old_ods</td>
<td>ODS versions before ODS@1 are not supported</td>
</tr>
</tbody>
</table>
### Appendix B: Exception Codes and Messages

<table>
<thead>
<tr>
<th>SQL-Code</th>
<th>GDSCODE</th>
<th>Symbol</th>
<th>Message Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>-806</td>
<td>335544600</td>
<td>col_name_err</td>
<td>Only simple column names permitted for VIEW WITH CHECK OPTION</td>
</tr>
<tr>
<td>-807</td>
<td>335544601</td>
<td>where_err</td>
<td>No WHERE clause for VIEW WITH CHECK OPTION</td>
</tr>
<tr>
<td>-808</td>
<td>335544602</td>
<td>table_view_err</td>
<td>Only one table allowed for VIEW WITH CHECK OPTION</td>
</tr>
<tr>
<td>-809</td>
<td>335544603</td>
<td>distinct_err</td>
<td>DISTINCT, GROUP or HAVING not permitted for VIEW WITH CHECK OPTION</td>
</tr>
<tr>
<td>-810</td>
<td>335544605</td>
<td>subquery_err</td>
<td>No subqueries permitted for VIEW WITH CHECK OPTION</td>
</tr>
<tr>
<td>-811</td>
<td>335544652</td>
<td>sing_select_err</td>
<td>multiple rows in singleton select</td>
</tr>
<tr>
<td>-816</td>
<td>335544651</td>
<td>ext_readonly_err</td>
<td>Cannot insert because the file is readonly or is on a read only medium.</td>
</tr>
<tr>
<td>-816</td>
<td>335544715</td>
<td>extfile_uns_op</td>
<td>Operation not supported for EXTERNAL FILE table @1</td>
</tr>
<tr>
<td>-817</td>
<td>335544361</td>
<td>read_only_trans</td>
<td>attempted update during read-only transaction</td>
</tr>
<tr>
<td>-817</td>
<td>335544371</td>
<td>segstr_no_write</td>
<td>attempted write to read-only BLOB</td>
</tr>
<tr>
<td>-817</td>
<td>335544444</td>
<td>read_only</td>
<td>operation not supported</td>
</tr>
<tr>
<td>-817</td>
<td>335544765</td>
<td>read_only_database</td>
<td>attempted update on read-only database</td>
</tr>
<tr>
<td>-817</td>
<td>335544766</td>
<td>must_be_dialect_2_and_up</td>
<td>SQL dialect @1 is not supported in this database</td>
</tr>
<tr>
<td>-817</td>
<td>335544793</td>
<td>ddl_not_allowed_by_db_sql_dial</td>
<td>Metadata update statement is not allowed by the current database SQL dialect @1</td>
</tr>
<tr>
<td>-817</td>
<td>336003079</td>
<td>sql_dialect_conflict_num</td>
<td>DB dialect @1 and client dialect @2 conflict with respect to numeric precision @3.</td>
</tr>
<tr>
<td>-817</td>
<td>336003101</td>
<td>upd_ins_with_complex_view</td>
<td>UPDATE OR INSERT without MATCHING could not be used with views based on more than one table</td>
</tr>
<tr>
<td>-817</td>
<td>336003102</td>
<td>dsq1_incompatible_trigger_type</td>
<td>Incompatible trigger type</td>
</tr>
<tr>
<td>-817</td>
<td>336003103</td>
<td>dsq1_db_trigger_type_cant_change</td>
<td>Database trigger type can't be changed</td>
</tr>
<tr>
<td>-820</td>
<td>335544356</td>
<td>obsolete_metadata</td>
<td>metadata is obsolete</td>
</tr>
<tr>
<td>-820</td>
<td>335544379</td>
<td>wrong_ods</td>
<td>unsupported on-disk structure for file @1; found @2.@3, support @4.@5</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-820</td>
<td>335544437</td>
<td>wrodynver</td>
<td>wrong DYN version</td>
</tr>
<tr>
<td>-820</td>
<td>335544467</td>
<td>high_minor</td>
<td>minor version too high found @1 expected @2</td>
</tr>
<tr>
<td>-820</td>
<td>335544881</td>
<td>need_difference</td>
<td>Difference file name should be set explicitly for database on raw device</td>
</tr>
<tr>
<td>-823</td>
<td>33554473</td>
<td>invalid_bookmark</td>
<td>invalid bookmark handle</td>
</tr>
<tr>
<td>-824</td>
<td>33554474</td>
<td>bad_lock_level</td>
<td>invalid lock level @1</td>
</tr>
<tr>
<td>-825</td>
<td>335544519</td>
<td>bad_lock_handle</td>
<td>invalid lock handle</td>
</tr>
<tr>
<td>-826</td>
<td>33554585</td>
<td>dsql_stmt_handle</td>
<td>Invalid statement handle</td>
</tr>
<tr>
<td>-827</td>
<td>335544555</td>
<td>invalid_direction</td>
<td>invalid direction for find operation</td>
</tr>
<tr>
<td>-827</td>
<td>335544718</td>
<td>invalid_key</td>
<td>Invalid key for find operation</td>
</tr>
<tr>
<td>-828</td>
<td>335544678</td>
<td>inval_key_posn</td>
<td>invalid key position</td>
</tr>
<tr>
<td>-829</td>
<td>335544616</td>
<td>field_ref_err</td>
<td>invalid column reference</td>
</tr>
<tr>
<td>-829</td>
<td>33606816</td>
<td>dyn_char_fld_too_small</td>
<td>New size specified for column @1 must be at least @2 characters.</td>
</tr>
<tr>
<td>-829</td>
<td>33606817</td>
<td>dyn_invalid_dtype_conversion</td>
<td>Cannot change datatype for @1. Conversion from base type @2 to @3 is not supported.</td>
</tr>
<tr>
<td>-829</td>
<td>33606818</td>
<td>dyn_drtype_conv_invalid</td>
<td>Cannot change datatype for column @1 from a character type to a non-character type.</td>
</tr>
<tr>
<td>-829</td>
<td>33606829</td>
<td>max_coll_per_charset</td>
<td>Maximum number of collations per character set exceeded</td>
</tr>
<tr>
<td>-829</td>
<td>33606830</td>
<td>invalid_coll_attr</td>
<td>Invalid collation attributes</td>
</tr>
<tr>
<td>-829</td>
<td>33606852</td>
<td>dyn_scale_too_big</td>
<td>New scale specified for column @1 must be at most @2.</td>
</tr>
<tr>
<td>-829</td>
<td>33606853</td>
<td>dyn_precision_too_small</td>
<td>New precision specified for column @1 must be at least @2.</td>
</tr>
<tr>
<td>-829</td>
<td>33606857</td>
<td>dyn_cannot_addrem_computed</td>
<td>Cannot add or remove COMPUTED from column @1</td>
</tr>
<tr>
<td>-830</td>
<td>335544615</td>
<td>field_aggregate_err</td>
<td>column used with aggregate</td>
</tr>
<tr>
<td>-831</td>
<td>33554548</td>
<td>primary_key_exists</td>
<td>Attempt to define a second PRIMARY KEY for the same table</td>
</tr>
<tr>
<td>-832</td>
<td>33554604</td>
<td>key_field_count_err</td>
<td>FOREIGN KEY column count does not match PRIMARY KEY</td>
</tr>
<tr>
<td>-833</td>
<td>33554606</td>
<td>expression_eval_err</td>
<td>expression evaluation not supported</td>
</tr>
<tr>
<td>SQL CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-833</td>
<td>335544810</td>
<td>date_range_exceeded</td>
<td>value exceeds the range for valid dates</td>
</tr>
<tr>
<td>-833</td>
<td>335544912</td>
<td>time_range_exceeded</td>
<td>value exceeds the range for a valid time</td>
</tr>
<tr>
<td>-833</td>
<td>335544913</td>
<td>datetime_range_exceeded</td>
<td>value exceeds the range for valid timestamps</td>
</tr>
<tr>
<td>-833</td>
<td>335544937</td>
<td>invalid_type_datetime_op</td>
<td>Invalid data type in DATE/TIME/TIMESTAMP addition or subtraction in add_datettime()</td>
</tr>
<tr>
<td>-833</td>
<td>335544938</td>
<td>onlycan_add_timetodate</td>
<td>Only a TIME value can be added to a DATE value</td>
</tr>
<tr>
<td>-833</td>
<td>335544939</td>
<td>onlycan_add_datetotime</td>
<td>Only a DATE value can be added to a TIME value</td>
</tr>
<tr>
<td>-833</td>
<td>335544940</td>
<td>onlycansub_tstampfromtstamp</td>
<td>TIMESTAMP values can be subtracted only from another TIMESTAMP value</td>
</tr>
<tr>
<td>-833</td>
<td>335544941</td>
<td>onlyoneop_mustbe_tstamp</td>
<td>Only one operand can be of type TIMESTAMP</td>
</tr>
<tr>
<td>-833</td>
<td>335544942</td>
<td>invalid_extractpart_time</td>
<td>Only HOUR, MINUTE, SECOND and MILLISECOND can be extracted from TIME values</td>
</tr>
<tr>
<td>-833</td>
<td>335544943</td>
<td>invalid_extractpart_date</td>
<td>HOUR, MINUTE, SECOND and MILLISECOND cannot be extracted from DATE values</td>
</tr>
<tr>
<td>-833</td>
<td>335544944</td>
<td>invalidarg_extract</td>
<td>Invalid argument for EXTRACT() not being of DATE/TIME/TIMESTAMP type</td>
</tr>
<tr>
<td>-833</td>
<td>335544945</td>
<td>sysf_argmustbe_exact</td>
<td>Arguments for @1 must be integral types or NUMERIC/DECIMAL without scale</td>
</tr>
<tr>
<td>-833</td>
<td>335544946</td>
<td>sysf_argmustbe_exact_or_fp</td>
<td>First argument for @1 must be integral type or floating point type</td>
</tr>
<tr>
<td>-833</td>
<td>335544947</td>
<td>sysf_argviolates_uuidtype</td>
<td>Human readable UUID argument for @1 must be of string type</td>
</tr>
<tr>
<td>-833</td>
<td>335544948</td>
<td>sysf_argviolates_uuidlen</td>
<td>Human readable UUID argument for @2 must be of exact length @1</td>
</tr>
<tr>
<td>-833</td>
<td>335544949</td>
<td>sysf_argviolates_uuidfmt</td>
<td>Human readable UUID argument for @3 must have &quot;-&quot; at position @2 instead of &quot;@1&quot;</td>
</tr>
<tr>
<td>-833</td>
<td>335544950</td>
<td>sysf_argviolates_guidigits</td>
<td>Human readable UUID argument for @3 must have hex digit at position @2 instead of &quot;@1&quot;</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>--------</td>
<td>--------------</td>
</tr>
<tr>
<td>-833</td>
<td>335544951</td>
<td>sysf_invalid_addpart_time</td>
<td>Only HOUR, MINUTE, SECOND and MILLISECOND can be added to TIME values in @1</td>
</tr>
<tr>
<td>-833</td>
<td>335544952</td>
<td>sysf_invalid_add_datetime</td>
<td>Invalid data type in addition of part to DATE/TIME/TIMESTAMP in @1</td>
</tr>
<tr>
<td>-833</td>
<td>335544953</td>
<td>sysf_invalid_addpart_dtime</td>
<td>Invalid part @1 to be added to a DATE/TIME/TIMESTAMP value in @2</td>
</tr>
<tr>
<td>-833</td>
<td>335544954</td>
<td>sysf_invalid_add_dtime_rc</td>
<td>Expected DATE/TIME/TIMESTAMP type in evlDateAdd() result</td>
</tr>
<tr>
<td>-833</td>
<td>335544955</td>
<td>sysf_invalid_diff_dtime</td>
<td>Expected DATE/TIME/TIMESTAMP type as first and second argument to @1</td>
</tr>
<tr>
<td>-833</td>
<td>335544956</td>
<td>sysf_invalid_timediff</td>
<td>The result of TIME-&lt;value&gt; in @1 cannot be expressed in YEAR, MONTH, DAY or WEEK</td>
</tr>
<tr>
<td>-833</td>
<td>335544957</td>
<td>sysf_invalid_tstampdiff</td>
<td>The result of TIME-TIMESTAMP or TIMESTAMP-TIME in @1 cannot be expressed in HOUR, MINUTE, SECOND or MILLISECOND</td>
</tr>
<tr>
<td>-833</td>
<td>335544958</td>
<td>sysf_invalid_datetimediff</td>
<td>The result of DATE-TIME or TIME-DATE in @1 cannot be expressed in HOUR, MINUTE, SECOND and MILLISECOND</td>
</tr>
<tr>
<td>-833</td>
<td>335544959</td>
<td>sysf_invalid_diffpart</td>
<td>Invalid part @1 to express the difference between two DATE/TIME/TIMESTAMP values in @2</td>
</tr>
<tr>
<td>-833</td>
<td>335544960</td>
<td>sysf_argmustbe_positive</td>
<td>Argument for @1 must be positive</td>
</tr>
<tr>
<td>-833</td>
<td>335544961</td>
<td>sysf_basemustbe_positive</td>
<td>Base for @1 must be positive</td>
</tr>
<tr>
<td>-833</td>
<td>335544962</td>
<td>sysf_argnmustbe_nonneg</td>
<td>Argument #@1 for @2 must be zero or positive</td>
</tr>
<tr>
<td>-833</td>
<td>335544963</td>
<td>sysf_argnmustbe_positive</td>
<td>Argument #@1 for @2 must be positive</td>
</tr>
<tr>
<td>-833</td>
<td>335544964</td>
<td>sysf_invalid_zeropowneg</td>
<td>Base for @1 cannot be zero if exponent is negative</td>
</tr>
<tr>
<td>-833</td>
<td>335544965</td>
<td>sysf_invalid_negpowfp</td>
<td>Base for @1 cannot be negative if exponent is not an integral value</td>
</tr>
<tr>
<td>-833</td>
<td>335544966</td>
<td>sysf_invalid_scale</td>
<td>The numeric scale must be between -128 and 127 in @1</td>
</tr>
<tr>
<td>-833</td>
<td>335544967</td>
<td>sysf_argmustbe_nonneg</td>
<td>Argument for @1 must be zero or positive</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-833</td>
<td>335544968</td>
<td>sysf_binuuid_mustbe_str</td>
<td>Binary UUID argument for @1 must be of string type</td>
</tr>
<tr>
<td>-833</td>
<td>335544969</td>
<td>sysf_binuuid_wrongsize</td>
<td>Binary UUID argument for @2 must use @1 bytes</td>
</tr>
<tr>
<td>-833</td>
<td>335544976</td>
<td>sysf_argmustbe_nonzero</td>
<td>Argument for @1 must be different than zero</td>
</tr>
<tr>
<td>-833</td>
<td>335544977</td>
<td>sysf_argmustbe_range_inc1_1</td>
<td>Argument for @1 must be in the range [-1, 1]</td>
</tr>
<tr>
<td>-833</td>
<td>335544978</td>
<td>sysf_argmustbe_gteq_one</td>
<td>Argument for @1 must be greater or equal than one</td>
</tr>
<tr>
<td>-833</td>
<td>335544979</td>
<td>sysf_argmustbe_range_exc1_1</td>
<td>Argument for @1 must be in the range ]-1, 1[</td>
</tr>
<tr>
<td>-833</td>
<td>335544981</td>
<td>sys_f_p_overflow</td>
<td>Floating point overflow in built-in function @1</td>
</tr>
<tr>
<td>-833</td>
<td>335545009</td>
<td>sysf_invalid_trig_namespace</td>
<td>Invalid usage of context namespace DDL_TRIGGER</td>
</tr>
<tr>
<td>-833</td>
<td>335545024</td>
<td>sysf_argscant_both_be_zero</td>
<td>Arguments for @1 cannot both be zero</td>
</tr>
<tr>
<td>-833</td>
<td>335545046</td>
<td>max_args_exceeded</td>
<td>Maximum (@1) number of arguments exceeded for function @2</td>
</tr>
<tr>
<td>-833</td>
<td>336397240</td>
<td>dsql_eval_unknode</td>
<td>Unknown node type @1 in dsql/GEN_expr</td>
</tr>
<tr>
<td>-833</td>
<td>336397241</td>
<td>dsql_agg_wrongarg</td>
<td>Argument for @1 in dialect 1 must be string or numeric</td>
</tr>
<tr>
<td>-833</td>
<td>336397242</td>
<td>dsql_agg2_wrongarg</td>
<td>Argument for @1 in dialect 3 must be numeric</td>
</tr>
<tr>
<td>-833</td>
<td>336397243</td>
<td>dsql_nodateortime_pm_string</td>
<td>Strings cannot be added to or subtracted from DATE or TIME types</td>
</tr>
<tr>
<td>-833</td>
<td>336397244</td>
<td>dsql_invalid_datetime_subtract</td>
<td>Invalid data type for subtraction involving DATE, TIME or TIMESTAMP types</td>
</tr>
<tr>
<td>-833</td>
<td>336397245</td>
<td>dsql_invalid_dateortime_add</td>
<td>Adding two DATE values or two TIME values is not allowed</td>
</tr>
<tr>
<td>-833</td>
<td>336397246</td>
<td>dsql_invalid_type_minus_date</td>
<td>DATE value cannot be subtracted from the provided data type</td>
</tr>
<tr>
<td>-833</td>
<td>336397247</td>
<td>dsql_nostring_addsub_dial3</td>
<td>Strings cannot be added or subtracted in dialect 3</td>
</tr>
<tr>
<td>-833</td>
<td>336397248</td>
<td>dsql_invalid_type_addsub_dial3</td>
<td>Invalid data type for addition or subtraction in dialect 3</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-833</td>
<td>336397249</td>
<td>dsq_invalid_type_multip_dial1</td>
<td>Invalid data type for multiplication in dialect 1</td>
</tr>
<tr>
<td>-833</td>
<td>336397250</td>
<td>dsq_nostring_multip_dial3</td>
<td>Strings cannot be multiplied in dialect 3</td>
</tr>
<tr>
<td>-833</td>
<td>336397251</td>
<td>dsq_invalid_type_multip_dial3</td>
<td>Invalid data type for multiplication in dialect 3</td>
</tr>
<tr>
<td>-833</td>
<td>336397252</td>
<td>dsq_mustuse_numeric_div_dial1</td>
<td>Division in dialect 1 must be between numeric data types</td>
</tr>
<tr>
<td>-833</td>
<td>336397253</td>
<td>dsq_nostring_div_dial3</td>
<td>Strings cannot be divided in dialect 3</td>
</tr>
<tr>
<td>-833</td>
<td>336397254</td>
<td>dsq_invalid_type_div_dial3</td>
<td>Invalid data type for division in dialect 3</td>
</tr>
<tr>
<td>-833</td>
<td>336397255</td>
<td>dsq_nostring_neg_dial3</td>
<td>Strings cannot be negated (applied the minus operator) in dialect 3</td>
</tr>
<tr>
<td>-833</td>
<td>336397256</td>
<td>dsq_invalid_type_neg</td>
<td>Invalid data type for negation (minus operator)</td>
</tr>
<tr>
<td>-834</td>
<td>335544508</td>
<td>range_not_found</td>
<td>refresh range number @1 not found</td>
</tr>
<tr>
<td>-835</td>
<td>335544649</td>
<td>bad_checksum</td>
<td>bad checksum</td>
</tr>
<tr>
<td>-836</td>
<td>335544517</td>
<td>except</td>
<td>exception @1</td>
</tr>
<tr>
<td>-836</td>
<td>335544848</td>
<td>except2</td>
<td>exception @1</td>
</tr>
<tr>
<td>-836</td>
<td>335545016</td>
<td>formatted_exception</td>
<td>@1</td>
</tr>
<tr>
<td>-837</td>
<td>335544518</td>
<td>cache_restart</td>
<td>restart shared cache manager</td>
</tr>
<tr>
<td>-838</td>
<td>335544560</td>
<td>shutwarn</td>
<td>database @1 shutdown in @2 seconds</td>
</tr>
<tr>
<td>-839</td>
<td>335544686</td>
<td>jrn_format_err</td>
<td>journal file wrong format</td>
</tr>
<tr>
<td>-840</td>
<td>335544687</td>
<td>jrn_file_full</td>
<td>intermediate journal file full</td>
</tr>
<tr>
<td>-841</td>
<td>335544677</td>
<td>version_err</td>
<td>too many versions</td>
</tr>
<tr>
<td>-842</td>
<td>335544697</td>
<td>precision_err</td>
<td>Precision must be from 1 to 18</td>
</tr>
<tr>
<td>-842</td>
<td>335544698</td>
<td>scale_nogt</td>
<td>Scale must be between zero and precision</td>
</tr>
<tr>
<td>-842</td>
<td>335544699</td>
<td>expec_short</td>
<td>Short integer expected</td>
</tr>
<tr>
<td>-842</td>
<td>335544700</td>
<td>expec_long</td>
<td>Long integer expected</td>
</tr>
<tr>
<td>-842</td>
<td>335544701</td>
<td>expec_ushort</td>
<td>Unsigned short integer expected</td>
</tr>
<tr>
<td>-842</td>
<td>335544712</td>
<td>expec_positive</td>
<td>Positive value expected</td>
</tr>
<tr>
<td>-901</td>
<td>335544322</td>
<td>bad_dbkey</td>
<td>invalid database key</td>
</tr>
<tr>
<td>-901</td>
<td>335544326</td>
<td>bad_dpb_form</td>
<td>unrecognized database parameter block</td>
</tr>
<tr>
<td>-901</td>
<td>335544327</td>
<td>bad_req_handle</td>
<td>invalid request handle</td>
</tr>
<tr>
<td>-901</td>
<td>335544328</td>
<td>bad_segstr_handle</td>
<td>invalid BLOB handle</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>335544329</td>
<td>bad_segstr_id</td>
<td>invalid BLOB ID</td>
</tr>
<tr>
<td>-901</td>
<td>335544330</td>
<td>bad_tpb_content</td>
<td>invalid parameter in transaction parameter block</td>
</tr>
<tr>
<td>-901</td>
<td>335544331</td>
<td>bad_tpb_form</td>
<td>invalid format for transaction parameter block</td>
</tr>
<tr>
<td>-901</td>
<td>335544332</td>
<td>bad_trans_handle</td>
<td>invalid transaction handle (expecting explicit transaction start)</td>
</tr>
<tr>
<td>-901</td>
<td>335544337</td>
<td>excess_trans</td>
<td>attempt to start more than @1 transactions</td>
</tr>
<tr>
<td>-901</td>
<td>335544339</td>
<td>infinap</td>
<td>information type inappropriate for object specified</td>
</tr>
<tr>
<td>-901</td>
<td>335544340</td>
<td>infona</td>
<td>no information of this type available for object specified</td>
</tr>
<tr>
<td>-901</td>
<td>335544341</td>
<td>infunk</td>
<td>unknown information item</td>
</tr>
<tr>
<td>-901</td>
<td>335544342</td>
<td>integ_fail</td>
<td>action cancelled by trigger (@1) to preserve data integrity</td>
</tr>
<tr>
<td>-901</td>
<td>335544345</td>
<td>lock_conflict</td>
<td>lock conflict on no wait transaction</td>
</tr>
<tr>
<td>-901</td>
<td>335544350</td>
<td>no_finish</td>
<td>program attempted to exit without finishing database</td>
</tr>
<tr>
<td>-901</td>
<td>335544353</td>
<td>no_recon</td>
<td>transaction is not in limbo</td>
</tr>
<tr>
<td>-901</td>
<td>335544355</td>
<td>no_segstr_close</td>
<td>BLOB was not closed</td>
</tr>
<tr>
<td>-901</td>
<td>335544357</td>
<td>open_trans</td>
<td>cannot disconnect database with open transactions (@1 active)</td>
</tr>
<tr>
<td>-901</td>
<td>335544358</td>
<td>port_len</td>
<td>message length error (encountered @1, expected @2)</td>
</tr>
<tr>
<td>-901</td>
<td>335544363</td>
<td>req_no_trans</td>
<td>no transaction for request</td>
</tr>
<tr>
<td>-901</td>
<td>335544364</td>
<td>req_sync</td>
<td>request synchronization error</td>
</tr>
<tr>
<td>-901</td>
<td>335544365</td>
<td>req_wrong_db</td>
<td>request referenced an unavailable database</td>
</tr>
<tr>
<td>-901</td>
<td>335544369</td>
<td>segstr_no_read</td>
<td>attempted read of a new, open BLOB</td>
</tr>
<tr>
<td>-901</td>
<td>335544370</td>
<td>segstr_no_trans</td>
<td>attempted action on BLOB outside transaction</td>
</tr>
<tr>
<td>-901</td>
<td>335544372</td>
<td>segstr_wrong_db</td>
<td>attempted reference to BLOB in unavailable database</td>
</tr>
<tr>
<td>-901</td>
<td>335544376</td>
<td>unres_rel</td>
<td>table @1 was omitted from the transaction reserving list</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>-----------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>335544377</td>
<td>uns_ext</td>
<td>request includes a DSRI extension not supported in this implementation</td>
</tr>
<tr>
<td>-901</td>
<td>335544378</td>
<td>wish_list</td>
<td>feature is not supported</td>
</tr>
<tr>
<td>-901</td>
<td>335544382</td>
<td>random</td>
<td>@1</td>
</tr>
<tr>
<td>-901</td>
<td>335544383</td>
<td>fatal_conflict</td>
<td>unrecoverable conflict with limbo transaction @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544392</td>
<td>bdbincon</td>
<td>internal error</td>
</tr>
<tr>
<td>-901</td>
<td>335544407</td>
<td>dbbnotzer</td>
<td>database handle not zero</td>
</tr>
<tr>
<td>-901</td>
<td>335544408</td>
<td>tranotzer</td>
<td>transaction handle not zero</td>
</tr>
<tr>
<td>-901</td>
<td>335544418</td>
<td>trainlim</td>
<td>transaction in limbo</td>
</tr>
<tr>
<td>-901</td>
<td>335544419</td>
<td>notinlim</td>
<td>transaction not in limbo</td>
</tr>
<tr>
<td>-901</td>
<td>335544420</td>
<td>traoutsta</td>
<td>transaction outstanding</td>
</tr>
<tr>
<td>-901</td>
<td>335544428</td>
<td>badmsgnum</td>
<td>undefined message number</td>
</tr>
<tr>
<td>-901</td>
<td>335544431</td>
<td>blocking_signal</td>
<td>blocking signal has been received</td>
</tr>
<tr>
<td>-901</td>
<td>335544442</td>
<td>noargacc_read</td>
<td>database system cannot read argument @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544443</td>
<td>noargacc_write</td>
<td>database system cannot write argument @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544450</td>
<td>misc_interpreted</td>
<td>@1</td>
</tr>
<tr>
<td>-901</td>
<td>335544468</td>
<td>tra_state</td>
<td>transaction @1 is @2</td>
</tr>
<tr>
<td>-901</td>
<td>335544485</td>
<td>bad_stmt_handle</td>
<td>invalid statement handle</td>
</tr>
<tr>
<td>-901</td>
<td>335544510</td>
<td>lock_timeout</td>
<td>lock time-out on wait transaction</td>
</tr>
<tr>
<td>-901</td>
<td>335544559</td>
<td>bad_svc_handle</td>
<td>invalid service handle</td>
</tr>
<tr>
<td>-901</td>
<td>335544561</td>
<td>wrospbver</td>
<td>wrong version of service parameter block</td>
</tr>
<tr>
<td>-901</td>
<td>335544562</td>
<td>bad_spb_form</td>
<td>unrecognized service parameter block</td>
</tr>
<tr>
<td>-901</td>
<td>335544563</td>
<td>svcnotdef</td>
<td>service @1 is not defined</td>
</tr>
<tr>
<td>-901</td>
<td>335544609</td>
<td>index_name</td>
<td>INDEX @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544610</td>
<td>exception_name</td>
<td>EXCEPTION @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544611</td>
<td>field_name</td>
<td>COLUMN @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544613</td>
<td>union_err</td>
<td>union not supported</td>
</tr>
<tr>
<td>-901</td>
<td>335544614</td>
<td>dsql_construct_err</td>
<td>Unsupported DSQL construct</td>
</tr>
<tr>
<td>-901</td>
<td>335544623</td>
<td>dsql_domain_err</td>
<td>Illegal use of keyword VALUE</td>
</tr>
<tr>
<td>-901</td>
<td>335544626</td>
<td>table_name</td>
<td>TABLE @1</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>335544627</td>
<td>proc_name</td>
<td>PROCEDURE @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544641</td>
<td>dsqldomain_not_found</td>
<td>Specified domain or source column @1 does not exist</td>
</tr>
<tr>
<td>-901</td>
<td>335544656</td>
<td>dsqldvar_conflict</td>
<td>variable @1 conflicts with parameter in same procedure</td>
</tr>
<tr>
<td>-901</td>
<td>335544666</td>
<td>srvr_version_too_old</td>
<td>server version too old to support all CREATE DATABASE options</td>
</tr>
<tr>
<td>-901</td>
<td>335544673</td>
<td>no_delete</td>
<td>cannot delete</td>
</tr>
<tr>
<td>-901</td>
<td>335544675</td>
<td>sort_err</td>
<td>sort error</td>
</tr>
<tr>
<td>-901</td>
<td>335544703</td>
<td>svcnoexe</td>
<td>service @1 does not have an associated executable</td>
</tr>
<tr>
<td>-901</td>
<td>335544704</td>
<td>net_lookup_err</td>
<td>Failed to locate host machine.</td>
</tr>
<tr>
<td>-901</td>
<td>335544705</td>
<td>service_unknown</td>
<td>Undefined service @1/@2.</td>
</tr>
<tr>
<td>-901</td>
<td>335544706</td>
<td>host_unknown</td>
<td>The specified name was not found in the hosts file or Domain Name Services.</td>
</tr>
<tr>
<td>-901</td>
<td>335544711</td>
<td>unprepared_stmt</td>
<td>Attempt to execute an unprepared dynamic SQL statement.</td>
</tr>
<tr>
<td>-901</td>
<td>335544716</td>
<td>svc_in_use</td>
<td>Service is currently busy: @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544719</td>
<td>net_init_error</td>
<td>Error initializing the network software.</td>
</tr>
<tr>
<td>-901</td>
<td>335544720</td>
<td>loadlib_failure</td>
<td>Unable to load required library @1.</td>
</tr>
<tr>
<td>-901</td>
<td>335544731</td>
<td>tra_must_sweep</td>
<td></td>
</tr>
<tr>
<td>-901</td>
<td>335544740</td>
<td>udf_exception</td>
<td>A fatal exception occurred during the execution of a user defined function.</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>---------</td>
<td>----------------</td>
<td>----------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>335544771</td>
<td>exception_float_denormal_operand</td>
<td>Float denormal operand. One of the floating-point operands is too small to represent a standard float value.</td>
</tr>
<tr>
<td>-901</td>
<td>335544772</td>
<td>exception_float_divide_by_zero</td>
<td>Floating-point divide by zero. The code attempted to divide a floating-point value by zero.</td>
</tr>
<tr>
<td>-901</td>
<td>335544773</td>
<td>exception_float_inexact_result</td>
<td>Floating-point inexact result. The result of a floating-point operation cannot be represented as a decimal fraction.</td>
</tr>
<tr>
<td>-901</td>
<td>335544774</td>
<td>exception_float_invalid_operand</td>
<td>Floating-point invalid operand. An indeterminant error occurred during a floating-point operation.</td>
</tr>
<tr>
<td>-901</td>
<td>335544775</td>
<td>exception_float_overflow</td>
<td>Floating-point overflow. The exponent of a floating-point operation is greater than the magnitude allowed.</td>
</tr>
<tr>
<td>-901</td>
<td>335544776</td>
<td>exception_float_stack_check</td>
<td>Floating-point stack check. The stack overflowed or underflowed as the result of a floating-point operation.</td>
</tr>
<tr>
<td>-901</td>
<td>335544777</td>
<td>exception_float_underflow</td>
<td>Floating-point underflow. The exponent of a floating-point operation is less than the magnitude allowed.</td>
</tr>
<tr>
<td>-901</td>
<td>335544778</td>
<td>exception_integer_divide_by_zero</td>
<td>Integer divide by zero. The code attempted to divide an integer value by an integer divisor of zero.</td>
</tr>
<tr>
<td>-901</td>
<td>335544779</td>
<td>exception_integer_overflow</td>
<td>Integer overflow. The result of an integer operation caused the most significant bit of the result to carry.</td>
</tr>
<tr>
<td>-901</td>
<td>335544780</td>
<td>exception_unknown</td>
<td>An exception occurred that does not have a description. Exception number @1.</td>
</tr>
<tr>
<td>-901</td>
<td>335544781</td>
<td>exception_stack_overflow</td>
<td>Stack overflow. The resource requirements of the runtime stack have exceeded the memory available to it.</td>
</tr>
<tr>
<td>-901</td>
<td>335544782</td>
<td>exception_sigsegv</td>
<td>Segmentation Fault. The code attempted to access memory without privileges.</td>
</tr>
<tr>
<td>-901</td>
<td>335544783</td>
<td>exception_sigill</td>
<td>Illegal Instruction. The Code attempted to perform an illegal operation.</td>
</tr>
<tr>
<td>-901</td>
<td>335544784</td>
<td>exception_sigbus</td>
<td>Bus Error. The Code caused a system bus error.</td>
</tr>
<tr>
<td>SQL- CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>--------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>335544785</td>
<td>exception_sigfpe</td>
<td>Floating Point Error. The Code caused an Arithmetic Exception or a floating point exception.</td>
</tr>
<tr>
<td>-901</td>
<td>335544786</td>
<td>ext_file_delete</td>
<td>Cannot delete rows from external files.</td>
</tr>
<tr>
<td>-901</td>
<td>335544787</td>
<td>ext_file_modify</td>
<td>Cannot update rows in external files.</td>
</tr>
<tr>
<td>-901</td>
<td>335544788</td>
<td>adm_task_denied</td>
<td>Unable to perform operation. You must be either SYSDBA or owner of the database</td>
</tr>
<tr>
<td>-901</td>
<td>335544794</td>
<td>cancelled</td>
<td>operation was cancelled</td>
</tr>
<tr>
<td>-901</td>
<td>335544797</td>
<td>svcnouser</td>
<td>user name and password are required while attaching to the services manager</td>
</tr>
<tr>
<td>-901</td>
<td>335544801</td>
<td>datatype_notsup</td>
<td>data type not supported for arithmetic</td>
</tr>
<tr>
<td>-901</td>
<td>335544803</td>
<td>dialect_not_changed</td>
<td>Database dialect not changed.</td>
</tr>
<tr>
<td>-901</td>
<td>335544804</td>
<td>database_create_failed</td>
<td>Unable to create database @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544805</td>
<td>inv_dialect_specified</td>
<td>Database dialect @1 is not a valid dialect.</td>
</tr>
<tr>
<td>-901</td>
<td>335544806</td>
<td>valid_db_dialects</td>
<td>Valid database dialects are @1.</td>
</tr>
<tr>
<td>-901</td>
<td>335544811</td>
<td>inv_client_dialect_specified</td>
<td>passed client dialect @1 is not a valid dialect.</td>
</tr>
<tr>
<td>-901</td>
<td>335544812</td>
<td>valid_client_dialects</td>
<td>Valid client dialects are @1.</td>
</tr>
<tr>
<td>-901</td>
<td>335544814</td>
<td>service_not_supported</td>
<td>Services functionality will be supported in a later version of the product</td>
</tr>
<tr>
<td>-901</td>
<td>335544820</td>
<td>invalid_savepoint</td>
<td>Unable to find savepoint with name @1 in transaction context</td>
</tr>
<tr>
<td>-901</td>
<td>335544835</td>
<td>bad_shutdown_mode</td>
<td>Target shutdown mode is invalid for database &quot;@1&quot;</td>
</tr>
<tr>
<td>-901</td>
<td>335544840</td>
<td>no_update</td>
<td>cannot update</td>
</tr>
<tr>
<td>-901</td>
<td>335544842</td>
<td>stack_trace</td>
<td>@1</td>
</tr>
<tr>
<td>-901</td>
<td>335544843</td>
<td>ctx_var_not_found</td>
<td>Context variable @1 is not found in namespace @2</td>
</tr>
<tr>
<td>-901</td>
<td>335544844</td>
<td>ctx_namespace_invalid</td>
<td>Invalid namespace name @1 passed to @2</td>
</tr>
<tr>
<td>-901</td>
<td>335544845</td>
<td>ctx_too_big</td>
<td>Too many context variables</td>
</tr>
<tr>
<td>-901</td>
<td>335544846</td>
<td>ctx_bad_argument</td>
<td>Invalid argument passed to @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544847</td>
<td>identifier_too_long</td>
<td>BLR syntax error. Identifier @1... is too long</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>335544859</td>
<td>invalid_time_precision</td>
<td>Time precision exceeds allowed range (0-@1)</td>
</tr>
<tr>
<td>-901</td>
<td>335544866</td>
<td>met_wrong_gtt_scope</td>
<td>@1 cannot depend on @2</td>
</tr>
<tr>
<td>-901</td>
<td>335544868</td>
<td>illegal_prc_type</td>
<td>Procedure @1 is not selectable (it does not contain a SUSPEND statement)</td>
</tr>
<tr>
<td>-901</td>
<td>335544869</td>
<td>invalid_sort_datatype</td>
<td>Datatype @1 is not supported for sorting operation</td>
</tr>
<tr>
<td>-901</td>
<td>335544870</td>
<td>collation_name</td>
<td>COLLATION @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544871</td>
<td>domain_name</td>
<td>DOMAIN @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544874</td>
<td>max_db_per_trans_allowed</td>
<td>A multi database transaction cannot span more than @1 databases</td>
</tr>
<tr>
<td>-901</td>
<td>335544876</td>
<td>bad_proc_BLR</td>
<td>Error while parsing procedure @1’s BLR</td>
</tr>
<tr>
<td>-901</td>
<td>335544877</td>
<td>key_too_big</td>
<td>index key too big</td>
</tr>
<tr>
<td>-901</td>
<td>335544885</td>
<td>bad_teb_form</td>
<td>Invalid TEB format</td>
</tr>
<tr>
<td>-901</td>
<td>335544886</td>
<td>tpb_multiple_txn_isolation</td>
<td>Found more than one transaction isolation in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544887</td>
<td>tpb_reserv_before_table</td>
<td>Table reservation lock type @1 requires table name before in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544888</td>
<td>tpb_multiple_spec</td>
<td>Found more than one @1 specification in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544889</td>
<td>tpb_option_without_rc</td>
<td>Option @1 requires READ COMMITTED isolation in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544890</td>
<td>tpb_conflicting_options</td>
<td>Option @1 is not valid if @2 was used previously in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544891</td>
<td>tpb_reserv_missing_tlen</td>
<td>Table name length missing after table reservation @1 in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544892</td>
<td>tpb_reserv_long_tlen</td>
<td>Table name length @1 is too long after table reservation @2 in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544893</td>
<td>tpb_reserv_missing_tname</td>
<td>Table name length @1 without table name after table reservation @2 in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544894</td>
<td>tpb_reserv_corrupt_tlen</td>
<td>Table name length @1 goes beyond the remaining TPB size after table reservation @2</td>
</tr>
<tr>
<td>-901</td>
<td>335544895</td>
<td>tpb_reserv_null_tlen</td>
<td>Table name length is zero after table reservation @1 in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544896</td>
<td>tpb_reserv_relnotfound</td>
<td>Table or view @1 not defined in system tables after table reservation @2 in TPB</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>335544897</td>
<td>tpb_reserv_baserelnotfound</td>
<td>Base table or view @1 for view @2 not defined in system tables after table reservation @3 in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544898</td>
<td>tpb_missing_len</td>
<td>Option length missing after option @1 in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544899</td>
<td>tpb_missing_value</td>
<td>Option length @1 without value after option @2 in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544900</td>
<td>tpb_corrupt_len</td>
<td>Option length @1 goes beyond the remaining TPB size after option @2</td>
</tr>
<tr>
<td>-901</td>
<td>335544901</td>
<td>tpb_null_len</td>
<td>Option length is zero after table reservation @1 in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544902</td>
<td>tpb_overflow_len</td>
<td>Option length @1 exceeds the range for option @2 in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544903</td>
<td>tpb_invalid_value</td>
<td>Option value @1 is invalid for the option @2 in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544904</td>
<td>tpb_reserv_stronger_wng</td>
<td>Preserving previous table reservation @1 for table @2, stronger than new @3 in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544905</td>
<td>tpb_reserv_stronger</td>
<td>Table reservation @1 for table @2 already specified and is stronger than new @3 in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544906</td>
<td>tpb_reserv_max_recursion</td>
<td>Table reservation reached maximum recursion of @1 when expanding views in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544907</td>
<td>tpb_reserv_virtualtbl</td>
<td>Table reservation in TPB cannot be applied to @1 because it's a virtual table</td>
</tr>
<tr>
<td>-901</td>
<td>335544908</td>
<td>tpb_reserv_systbl</td>
<td>Table reservation in TPB cannot be applied to @1 because it's a system table</td>
</tr>
<tr>
<td>-901</td>
<td>335544909</td>
<td>tpb_reserv_temptbl</td>
<td>Table reservation @1 or @2 in TPB cannot be applied to @3 because it's a temporary table</td>
</tr>
<tr>
<td>-901</td>
<td>335544910</td>
<td>tpb_readtxn_after_writelock</td>
<td>Cannot set the transaction in read only mode after a table reservation isc_tpb_lock_write in TPB</td>
</tr>
<tr>
<td>-901</td>
<td>335544911</td>
<td>tpb_writelock_after_readtxn</td>
<td>Cannot take a table reservation isc_tpb_lock_write in TPB because the transaction is in read only mode</td>
</tr>
<tr>
<td>-901</td>
<td>335544917</td>
<td>shutdown_timeout</td>
<td>Firebird shutdown is still in progress after the specified timeout</td>
</tr>
<tr>
<td>SQL- CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>335544918</td>
<td>att_handle_busy</td>
<td>Attachment handle is busy</td>
</tr>
<tr>
<td>-901</td>
<td>335544919</td>
<td>bad_udf_freeit</td>
<td>Bad written UDF detected: pointer returned in FREE_IT function was not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>allocated by ib_util_malloc</td>
</tr>
<tr>
<td>-901</td>
<td>335544920</td>
<td>eds_provider_not_found</td>
<td>External Data Source provider '@1' not found</td>
</tr>
<tr>
<td>-901</td>
<td>335544921</td>
<td>eds_connection</td>
<td>Execute statement error at @1 : @2Data source : @3</td>
</tr>
<tr>
<td>-901</td>
<td>335544922</td>
<td>eds_preprocess</td>
<td>Execute statement preprocess SQL error</td>
</tr>
<tr>
<td>-901</td>
<td>335544923</td>
<td>eds_stmt_expected</td>
<td>Statement expected</td>
</tr>
<tr>
<td>-901</td>
<td>335544924</td>
<td>eds_prm_name_expected</td>
<td>Parameter name expected</td>
</tr>
<tr>
<td>-901</td>
<td>335544925</td>
<td>eds_unclosed_comment</td>
<td>Unclosed comment found near '@1'</td>
</tr>
<tr>
<td>-901</td>
<td>335544926</td>
<td>eds_statement</td>
<td>Execute statement error at @1 : @2Statement : @3 Data source : @4</td>
</tr>
<tr>
<td>-901</td>
<td>335544927</td>
<td>eds_input_prm_mismatch</td>
<td>Input parameters mismatch</td>
</tr>
<tr>
<td>-901</td>
<td>335544928</td>
<td>eds_output_prm_mismatch</td>
<td>Output parameters mismatch</td>
</tr>
<tr>
<td>-901</td>
<td>335544929</td>
<td>eds_input_prm_not_set</td>
<td>Input parameter '@1' have no value set</td>
</tr>
<tr>
<td>-901</td>
<td>335544933</td>
<td>nothing_to_cancel</td>
<td>nothing to cancel</td>
</tr>
<tr>
<td>-901</td>
<td>335544934</td>
<td>ibutil_not_loaded</td>
<td>ib_util library has not been loaded to deallocate memory returned by FREE_IT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>function</td>
</tr>
<tr>
<td>-901</td>
<td>335544973</td>
<td>bad_epb_form</td>
<td>Unrecognized events block</td>
</tr>
<tr>
<td>-901</td>
<td>335544982</td>
<td>udf_fp_overflow</td>
<td>Floating point overflow in result from UDF @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544983</td>
<td>udf_fp_nan</td>
<td>Invalid floating point value returned by UDF @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544985</td>
<td>out_of_temp_space</td>
<td>No free space found in temporary directories</td>
</tr>
<tr>
<td>-901</td>
<td>335544986</td>
<td>eds_expl_tran_ctrl</td>
<td>Explicit transaction control is not allowed</td>
</tr>
<tr>
<td>-901</td>
<td>335544988</td>
<td>package_name</td>
<td>PACKAGE @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544989</td>
<td>cannot_make_not_null</td>
<td>Cannot make field @1 of table @2 NOT NULL because there are NULLs present</td>
</tr>
<tr>
<td>-901</td>
<td>335544990</td>
<td>feature_removed</td>
<td>Feature @1 is not supported anymore</td>
</tr>
<tr>
<td>-901</td>
<td>335544991</td>
<td>view_name</td>
<td>VIEW @1</td>
</tr>
<tr>
<td>SQLCODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>335544993</td>
<td>invalid_fetch_option</td>
<td>Fetch option @1 is invalid for a non-scrollable cursor</td>
</tr>
<tr>
<td>-901</td>
<td>335544994</td>
<td>bad_fun_BLR</td>
<td>Error while parsing function @1’s BLR</td>
</tr>
<tr>
<td>-901</td>
<td>335544995</td>
<td>func_pack_not_implemented</td>
<td>Cannot execute function @1 of the unimplemented package @2</td>
</tr>
<tr>
<td>-901</td>
<td>335544996</td>
<td>proc_pack_not_implemented</td>
<td>Cannot execute procedure @1 of the unimplemented package @2</td>
</tr>
<tr>
<td>-901</td>
<td>335544997</td>
<td>eem_func_not_returned</td>
<td>External function @1 not returned by the external engine plugin @2</td>
</tr>
<tr>
<td>-901</td>
<td>335544998</td>
<td>eem_proc_not_returned</td>
<td>External procedure @1 not returned by the external engine plugin @2</td>
</tr>
<tr>
<td>-901</td>
<td>335544999</td>
<td>eem_trig_not_returned</td>
<td>External trigger @1 not returned by the external engine plugin @2</td>
</tr>
<tr>
<td>-901</td>
<td>335545000</td>
<td>eem_bad_plugin_ver</td>
<td>Incompatible plugin version @1 for external engine @2</td>
</tr>
<tr>
<td>-901</td>
<td>335545001</td>
<td>eem_engine_notfound</td>
<td>External engine @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>335545004</td>
<td>pman_cannot_load_plugin</td>
<td>Error loading plugin @1</td>
</tr>
<tr>
<td>-901</td>
<td>335545005</td>
<td>pman_module_notfound</td>
<td>Loadable module @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>335545006</td>
<td>pman_entrypoint_notfound</td>
<td>Standard plugin entrypoint does not exist in module @1</td>
</tr>
<tr>
<td>-901</td>
<td>335545007</td>
<td>pman_module_bad</td>
<td>Module @1 exists but can not be loaded</td>
</tr>
<tr>
<td>-901</td>
<td>335545008</td>
<td>pman_plugin_notfound</td>
<td>Module @1 does not contain plugin @2 type @3</td>
</tr>
<tr>
<td>-901</td>
<td>335545010</td>
<td>unexpected_null</td>
<td>Value is NULL but isNull parameter was not informed</td>
</tr>
<tr>
<td>-901</td>
<td>335545011</td>
<td>type_notcompat_blob</td>
<td>Type @1 is incompatible with BLOB</td>
</tr>
<tr>
<td>-901</td>
<td>335545012</td>
<td>invalid_date_val</td>
<td>Invalid date</td>
</tr>
<tr>
<td>-901</td>
<td>335545013</td>
<td>invalid_time_val</td>
<td>Invalid time</td>
</tr>
<tr>
<td>-901</td>
<td>335545014</td>
<td>invalid_timestamp_val</td>
<td>Invalid timestamp</td>
</tr>
<tr>
<td>-901</td>
<td>335545015</td>
<td>invalid_index_val</td>
<td>Invalid index @1 in function @2</td>
</tr>
<tr>
<td>-901</td>
<td>335545018</td>
<td>private_function</td>
<td>Function @1 is private to package @2</td>
</tr>
<tr>
<td>-901</td>
<td>335545019</td>
<td>private_procedure</td>
<td>Procedure @1 is private to package @2</td>
</tr>
<tr>
<td>-901</td>
<td>335545021</td>
<td>bad_events_handle</td>
<td>invalid events id (handle)</td>
</tr>
<tr>
<td>-901</td>
<td>335545025</td>
<td>spb_no_id</td>
<td>missing service ID in spb</td>
</tr>
<tr>
<td>-901</td>
<td>335545026</td>
<td>ee_blr_mismatch_null</td>
<td>External BLR message mismatch: invalid null descriptor at field @1</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>335545027</td>
<td>ee_blr_mismatch_length</td>
<td>External BLR message mismatch: length = @1, expected @2</td>
</tr>
<tr>
<td>-901</td>
<td>335545031</td>
<td>libtommath_generic</td>
<td>Libtommath error code @1 in function @2</td>
</tr>
<tr>
<td>-901</td>
<td>335545041</td>
<td>cp_process_active</td>
<td>Crypt failed - already crypting database</td>
</tr>
<tr>
<td>-901</td>
<td>335545042</td>
<td>cp_already_crypted</td>
<td>Crypt failed - database is already in requested state</td>
</tr>
<tr>
<td>-901</td>
<td>335545047</td>
<td>ee_blr_mismatch_names_count</td>
<td>External BLR message mismatch: names count = @1, blr count = @2</td>
</tr>
<tr>
<td>-901</td>
<td>335545048</td>
<td>ee_blr_mismatch_name_not_found</td>
<td>External BLR message mismatch: name @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>335545049</td>
<td>bad_result_set</td>
<td>Invalid resultset interface</td>
</tr>
<tr>
<td>-901</td>
<td>335545059</td>
<td>badvarnum</td>
<td>undefined variable number</td>
</tr>
<tr>
<td>-901</td>
<td>335545071</td>
<td>info_unprepared_stmt</td>
<td>Attempt to get information about an unprepared dynamic SQL statement.</td>
</tr>
<tr>
<td>-901</td>
<td>335545072</td>
<td>idx_key_value</td>
<td>Problematic key value is @1</td>
</tr>
<tr>
<td>-901</td>
<td>335545073</td>
<td>forupdate_virtualtbl</td>
<td>Cannot select virtual table @1 for update WITH LOCK</td>
</tr>
<tr>
<td>-901</td>
<td>335545074</td>
<td>forupdate_systbl</td>
<td>Cannot select system table @1 for update WITH LOCK</td>
</tr>
<tr>
<td>-901</td>
<td>335545075</td>
<td>forupdate_temptbl</td>
<td>Cannot select temporary table @1 for update WITH LOCK</td>
</tr>
<tr>
<td>-901</td>
<td>335545076</td>
<td>cant_modify_sysobj</td>
<td>System @1 @2 cannot be modified</td>
</tr>
<tr>
<td>-901</td>
<td>335545077</td>
<td>server_misconfigured</td>
<td>Server misconfigured - contact administrator please</td>
</tr>
<tr>
<td>-901</td>
<td>335545078</td>
<td>alter_role</td>
<td>Deprecated backward compatibility ALTER ROLE ... SET/DROP AUTO ADMIN mapping may be used only for RDB$ADMIN role</td>
</tr>
<tr>
<td>-901</td>
<td>335545079</td>
<td>map_already_exists</td>
<td>Mapping @1 already exists</td>
</tr>
<tr>
<td>-901</td>
<td>335545080</td>
<td>map_not_exists</td>
<td>Mapping @1 does not exist</td>
</tr>
<tr>
<td>-901</td>
<td>335545081</td>
<td>map_load</td>
<td>@1 failed when loading mapping cache</td>
</tr>
<tr>
<td>-901</td>
<td>335545082</td>
<td>map_aster</td>
<td>Invalid name &lt;*&gt; in authentication block</td>
</tr>
<tr>
<td>-901</td>
<td>335545083</td>
<td>map_multi</td>
<td>Multiple maps found for @1</td>
</tr>
<tr>
<td>-901</td>
<td>335545084</td>
<td>map_undefined</td>
<td>Undefined mapping result - more than one different results found</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>335545088</td>
<td>map_nodb</td>
<td>Global mapping is not available when database @1 is not present</td>
</tr>
<tr>
<td>-901</td>
<td>335545089</td>
<td>map_notable</td>
<td>Global mapping is not available when table RDB$MAP is not present in database @1</td>
</tr>
<tr>
<td>-901</td>
<td>335545090</td>
<td>miss_trusted_role</td>
<td>Your attachment has no trusted role</td>
</tr>
<tr>
<td>-901</td>
<td>335545091</td>
<td>set_invalid_role</td>
<td>Role @1 is invalid or unavailable</td>
</tr>
<tr>
<td>-901</td>
<td>335545093</td>
<td>dup_attribute</td>
<td>Duplicated user attribute @1</td>
</tr>
<tr>
<td>-901</td>
<td>335545094</td>
<td>dyn_no_priv</td>
<td>There is no privilege for this operation</td>
</tr>
<tr>
<td>-901</td>
<td>335545095</td>
<td>dsql_cant_grant_option</td>
<td>Using GRANT OPTION on @1 not allowed</td>
</tr>
<tr>
<td>-901</td>
<td>335545097</td>
<td>crdb_load</td>
<td>@1 failed when working with CREATE DATABASE grants</td>
</tr>
<tr>
<td>-901</td>
<td>335545098</td>
<td>crdb_nodb</td>
<td>CREATE DATABASE grants check is not possible when database @1 is not present</td>
</tr>
<tr>
<td>-901</td>
<td>335545099</td>
<td>crdb_notable</td>
<td>CREATE DATABASE grants check is not possible when table RDB$DB_CREATORS is not present in database @1</td>
</tr>
<tr>
<td>-901</td>
<td>335545102</td>
<td>savepoint_backout_err</td>
<td>Error during savepoint backout - transaction invalidated</td>
</tr>
<tr>
<td>-901</td>
<td>335545105</td>
<td>map_down</td>
<td>Some database(s) were shutdown when trying to read mapping data</td>
</tr>
<tr>
<td>-901</td>
<td>335545109</td>
<td>encrypt_error</td>
<td>Page requires encryption but crypt plugin is missing</td>
</tr>
<tr>
<td>-901</td>
<td>336068645</td>
<td>dyn_filter_not_found</td>
<td>BLOB Filter @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068649</td>
<td>dyn_func_not_found</td>
<td>Function @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068656</td>
<td>dyn_index_not_found</td>
<td>Index not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068662</td>
<td>dyn_view_not_found</td>
<td>View @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068697</td>
<td>dyn_domain_not_found</td>
<td>Domain not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068717</td>
<td>dyn_cant_modify_auto_trig</td>
<td>Triggers created automatically cannot be modified</td>
</tr>
<tr>
<td>-901</td>
<td>336068740</td>
<td>dyn_dup_table</td>
<td>Table @1 already exists</td>
</tr>
<tr>
<td>-901</td>
<td>336068748</td>
<td>dyn_proc_not_found</td>
<td>Procedure @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068752</td>
<td>dyn_exception_not_found</td>
<td>Exception not found</td>
</tr>
<tr>
<td>SQLCODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>--------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>336068754</td>
<td>dyn_proc_param_not_found</td>
<td>Parameter @1 in procedure @2 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068755</td>
<td>dyn_trig_not_found</td>
<td>Trigger @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068759</td>
<td>dyn_charset_not_found</td>
<td>Character set @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068760</td>
<td>dyn_collation_not_found</td>
<td>Collation @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068763</td>
<td>dyn_role_not_found</td>
<td>Role @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068767</td>
<td>dyn_name_longer</td>
<td>Name longer than database column size</td>
</tr>
<tr>
<td>-901</td>
<td>336068784</td>
<td>dyn_column_does_not_exist</td>
<td>Column @1 does not exist in table/view @2</td>
</tr>
<tr>
<td>-901</td>
<td>336068796</td>
<td>dyn_role_does_not_exist</td>
<td>SQL role @1 does not exist</td>
</tr>
<tr>
<td>-901</td>
<td>336068797</td>
<td>dyn_no_grant_admin_opt</td>
<td>User @1 has no grant admin option on SQL role @2</td>
</tr>
<tr>
<td>-901</td>
<td>336068798</td>
<td>dyn_user_not_role_member</td>
<td>User @1 is not a member of SQL role @2</td>
</tr>
<tr>
<td>-901</td>
<td>336068799</td>
<td>dyn_delete_role_failed</td>
<td>@1 is not the owner of SQL role @2</td>
</tr>
<tr>
<td>-901</td>
<td>336068800</td>
<td>dyn_grant_role_to_user</td>
<td>@1 is a SQL role and not a user</td>
</tr>
<tr>
<td>-901</td>
<td>336068801</td>
<td>dyn_inv_sql_role_name</td>
<td>User name @1 could not be used for SQL role</td>
</tr>
<tr>
<td>-901</td>
<td>336068802</td>
<td>dyn_dup_sql_role</td>
<td>SQL role @1 already exists</td>
</tr>
<tr>
<td>-901</td>
<td>336068803</td>
<td>dyn_kywd_spec_for_role</td>
<td>Keyword @1 can not be used as a SQL role name</td>
</tr>
<tr>
<td>-901</td>
<td>336068804</td>
<td>dyn_roles_not_supported</td>
<td>SQL roles are not supported in on older versions of the database. A backup and restore of the database is required.</td>
</tr>
<tr>
<td>-901</td>
<td>336068820</td>
<td>dyn_zero_len_id</td>
<td>Zero length identifiers are not allowed</td>
</tr>
<tr>
<td>-901</td>
<td>336068822</td>
<td>dyn_gen_not_found</td>
<td>Sequence @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068840</td>
<td>dyn_wrong_gtt_scope</td>
<td>@1 cannot reference @2</td>
</tr>
<tr>
<td>-901</td>
<td>336068843</td>
<td>dyn_coll_used_table</td>
<td>Collation @1 is used in table @2 (field name @3) and cannot be dropped</td>
</tr>
<tr>
<td>-901</td>
<td>336068844</td>
<td>dyn_coll_used_domain</td>
<td>Collation @1 is used in domain @2 and cannot be dropped</td>
</tr>
<tr>
<td>-901</td>
<td>336068846</td>
<td>dyn_cannot_del_def_coll</td>
<td>Cannot delete default collation of CHARACTER SET @1</td>
</tr>
<tr>
<td>-901</td>
<td>336068849</td>
<td>dyn_table_not_found</td>
<td>Table @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068851</td>
<td>dyn_coll_used_procedure</td>
<td>Collation @1 is used in procedure @2 (parameter name @3) and cannot be dropped</td>
</tr>
<tr>
<td>SQL-Code</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>----------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>336068856</td>
<td>dyn_ods_not_supp_feature</td>
<td>Feature '@1' is not supported in ODS @2.@3</td>
</tr>
<tr>
<td>-901</td>
<td>336068858</td>
<td>dyn_no_empty_pw</td>
<td>Password should not be empty string</td>
</tr>
<tr>
<td>-901</td>
<td>336068859</td>
<td>dyn_dup_index</td>
<td>Index @1 already exists</td>
</tr>
<tr>
<td>-901</td>
<td>336068864</td>
<td>dyn_package_not_found</td>
<td>Package @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068865</td>
<td>dyn_schema_not_found</td>
<td>Schema @1 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068871</td>
<td>dyn_funcnotdef_package</td>
<td>Function @1 has not been defined on the package body @2</td>
</tr>
<tr>
<td>-901</td>
<td>336068872</td>
<td>dyn_procnotdef_package</td>
<td>Procedure @1 has not been defined on the package body @2</td>
</tr>
<tr>
<td>-901</td>
<td>336068873</td>
<td>dyn_funcsignat_package</td>
<td>Function @1 has a signature mismatch on package body @2</td>
</tr>
<tr>
<td>-901</td>
<td>336068874</td>
<td>dyn_procsignat_package</td>
<td>Procedure @1 has a signature mismatch on package body @2</td>
</tr>
<tr>
<td>-901</td>
<td>336068875</td>
<td>dyn_defvaldecl_package_proc</td>
<td>Default values for parameters are allowed only in declaration of packaged procedure @1.@2</td>
</tr>
<tr>
<td>-901</td>
<td>336068877</td>
<td>dyn_package_body_exists</td>
<td>Package body @1 already exists</td>
</tr>
<tr>
<td>-901</td>
<td>336068879</td>
<td>dyn_newfc_oldsyntax</td>
<td>Cannot alter new style function @1 with ALTER EXTERNAL FUNCTION. Use ALTER FUNCTION instead.</td>
</tr>
<tr>
<td>-901</td>
<td>336068886</td>
<td>dyn_func_param_not_found</td>
<td>Parameter @1 in function @2 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068887</td>
<td>dyn_routine_param_not_found</td>
<td>Parameter @1 of routine @2 not found</td>
</tr>
<tr>
<td>-901</td>
<td>336068888</td>
<td>dyn_routine_param_ambiguous</td>
<td>Parameter @1 of routine @2 is ambiguous (found in both procedures and functions). Use a specifier keyword.</td>
</tr>
<tr>
<td>-901</td>
<td>336068889</td>
<td>dyn_coll_used_function</td>
<td>Collation @1 is used in function @2 (parameter name @3) and cannot be dropped</td>
</tr>
<tr>
<td>-901</td>
<td>336068890</td>
<td>dyn_domain_used_function</td>
<td>Domain @1 is used in function @2 (parameter name @3) and cannot be dropped</td>
</tr>
<tr>
<td>-901</td>
<td>336068891</td>
<td>dyn_alter_user_no_clause</td>
<td>ALTER USER requires at least one clause to be specified</td>
</tr>
<tr>
<td>-901</td>
<td>336068894</td>
<td>dyn_duplicate_package_item</td>
<td>Duplicate @1 @2</td>
</tr>
<tr>
<td>-901</td>
<td>336068895</td>
<td>dyn_cant_modify_sysobj</td>
<td>System @1 @2 cannot be modified</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>336068896</td>
<td>dyn_cant_use_zero_increment</td>
<td>INCREMENT BY 0 is an illegal option for sequence @1</td>
</tr>
<tr>
<td>-901</td>
<td>336068897</td>
<td>dyn_cant_use_in_foreignkey</td>
<td>Can’t use @1 in FOREIGN KEY constraint</td>
</tr>
<tr>
<td>-901</td>
<td>336068898</td>
<td>dyn_defvaldecl_package_func</td>
<td>Default values for parameters are allowed only in declaration of packaged function @1.@2</td>
</tr>
<tr>
<td>-901</td>
<td>336397211</td>
<td>dsq_too_many_values</td>
<td>Too many values (more than @1) in member list to match against</td>
</tr>
<tr>
<td>-901</td>
<td>336397236</td>
<td>dsq_unsupp_feature_dialect</td>
<td>feature is not supported in dialect @1</td>
</tr>
<tr>
<td>-901</td>
<td>336397239</td>
<td>dsq_unsupported_in_auto_trans</td>
<td>@1 is not supported inside IN AUTONOMOUS TRANSACTION block</td>
</tr>
<tr>
<td>-901</td>
<td>336397258</td>
<td>dsq_alter_charset_failed</td>
<td>ALTER CHARACTER SET @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397259</td>
<td>dsq_comment_on_failed</td>
<td>COMMENT ON @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397260</td>
<td>dsq_create_func_failed</td>
<td>CREATE FUNCTION @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397261</td>
<td>dsq_alter_func_failed</td>
<td>ALTER FUNCTION @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397262</td>
<td>dsq_create_alter_func_failed</td>
<td>CREATE OR ALTER FUNCTION @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397263</td>
<td>dsq_drop_func_failed</td>
<td>DROP FUNCTION @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397264</td>
<td>dsq_recreate_func_failed</td>
<td>RECREATE FUNCTION @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397265</td>
<td>dsq_create_proc_failed</td>
<td>CREATE PROCEDURE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397266</td>
<td>dsq_alter_proc_failed</td>
<td>ALTER PROCEDURE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397267</td>
<td>dsq_create_alter_proc_failed</td>
<td>CREATE OR ALTER PROCEDURE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397268</td>
<td>dsq_drop_proc_failed</td>
<td>DROP PROCEDURE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397269</td>
<td>dsq_recreate_proc_failed</td>
<td>RECREATE PROCEDURE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397270</td>
<td>dsq_create_trigger_failed</td>
<td>CREATE TRIGGER @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397271</td>
<td>dsq_alter_trigger_failed</td>
<td>ALTER TRIGGER @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397272</td>
<td>dsq_create_alter_trigger_failed</td>
<td>CREATE OR ALTER TRIGGER @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397273</td>
<td>dsq_drop_trigger_failed</td>
<td>DROP TRIGGER @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397274</td>
<td>dsq_recreate_trigger_failed</td>
<td>RECREATE TRIGGER @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397275</td>
<td>dsq_create_collation_failed</td>
<td>CREATE COLLATION @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397276</td>
<td>dsq_drop_collation_failed</td>
<td>DROP COLLATION @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397277</td>
<td>dsq_create_domain_failed</td>
<td>CREATE DOMAIN @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397278</td>
<td>dsq_alter_domain_failed</td>
<td>ALTER DOMAIN @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397279</td>
<td>dsq_drop_domain_failed</td>
<td>DROP DOMAIN @1 failed</td>
</tr>
</tbody>
</table>
## Appendix B: Exception Codes and Messages

<table>
<thead>
<tr>
<th>SQL-CODE</th>
<th>GDSCODE</th>
<th>Symbol</th>
<th>Message Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>-901</td>
<td>336397280</td>
<td>dsql_create_except_failed</td>
<td>CREATE EXCEPTION @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397281</td>
<td>dsql_alter_except_failed</td>
<td>ALTER EXCEPTION @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397282</td>
<td>dsql_create_alter_except_failed</td>
<td>CREATE OR ALTER EXCEPTION @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397283</td>
<td>dsql_recreate_except_failed</td>
<td>RECREATE EXCEPTION @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397284</td>
<td>dsql_drop_except_failed</td>
<td>DROP EXCEPTION @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397285</td>
<td>dsql_create_sequence_failed</td>
<td>CREATE SEQUENCE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397286</td>
<td>dsql_create_table_failed</td>
<td>CREATE TABLE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397287</td>
<td>dsql_alter_table_failed</td>
<td>ALTER TABLE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397288</td>
<td>dsql_drop_table_failed</td>
<td>DROP TABLE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397289</td>
<td>dsql_recreate_table_failed</td>
<td>RECREATE TABLE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397290</td>
<td>dsql_create_pack_failed</td>
<td>CREATE PACKAGE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397291</td>
<td>dsql_alter_pack_failed</td>
<td>ALTER PACKAGE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397292</td>
<td>dsql_create_alter_pack_failed</td>
<td>CREATE OR ALTER PACKAGE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397293</td>
<td>dsql_drop_pack_failed</td>
<td>DROP PACKAGE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397294</td>
<td>dsql_recreate_pack_failed</td>
<td>RECREATE PACKAGE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397295</td>
<td>dsql_create_pack_body_failed</td>
<td>CREATE PACKAGE BODY @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397296</td>
<td>dsql_drop_pack_body_failed</td>
<td>DROP PACKAGE BODY @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397297</td>
<td>dsql_recreate_pack_body_failed</td>
<td>RECREATE PACKAGE BODY @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397298</td>
<td>dsql_create_view_failed</td>
<td>CREATE VIEW @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397299</td>
<td>dsql_alter_view_failed</td>
<td>ALTER VIEW @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397300</td>
<td>dsql_create_alter_view_failed</td>
<td>CREATE OR ALTER VIEW @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397301</td>
<td>dsql_recreate_view_failed</td>
<td>RECREATE VIEW @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397302</td>
<td>dsql_drop_view_failed</td>
<td>DROP VIEW @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397303</td>
<td>dsql_drop_sequence_failed</td>
<td>DROP SEQUENCE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397304</td>
<td>dsql_recreate_sequence_failed</td>
<td>RECREATE SEQUENCE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397305</td>
<td>dsql_drop_index_failed</td>
<td>DROP INDEX @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397306</td>
<td>dsql_drop_filter_failed</td>
<td>DROP FILTER @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397307</td>
<td>dsql_drop_shadow_failed</td>
<td>DROP SHADOW @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397308</td>
<td>dsql_drop_role_failed</td>
<td>DROP ROLE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397309</td>
<td>dsql_drop_user_failed</td>
<td>DROP USER @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397310</td>
<td>dsql_create_role_failed</td>
<td>CREATE ROLE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397311</td>
<td>dsql_alter_role_failed</td>
<td>ALTER ROLE @1 failed</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>336397312</td>
<td>dsqalter_index_failed</td>
<td>ALTER INDEX @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397313</td>
<td>dsqalter_database_failed</td>
<td>ALTER DATABASE failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397314</td>
<td>dsq_create_shadow_failed</td>
<td>CREATE SHADOW @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397315</td>
<td>dsq_create_filter_failed</td>
<td>DECLARE FILTER @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397316</td>
<td>dsq_create_index_failed</td>
<td>CREATE INDEX @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397317</td>
<td>dsq_create_user_failed</td>
<td>CREATE USER @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397318</td>
<td>dsqalter_user_failed</td>
<td>ALTER USER @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397319</td>
<td>dsq_grant_failed</td>
<td>GRANT failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397320</td>
<td>dsq_revoke_failed</td>
<td>REVOKE failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397322</td>
<td>dsq_mapping_failed</td>
<td>@2 MAPPING @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397323</td>
<td>dsqalter_sequence_failed</td>
<td>ALTER SEQUENCE @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397324</td>
<td>dsq_create_generator_failed</td>
<td>CREATE GENERATOR @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397325</td>
<td>dsq_set_generator_failed</td>
<td>SET GENERATOR @1 failed</td>
</tr>
<tr>
<td>-901</td>
<td>336397330</td>
<td>dsq_max_exception_arguments</td>
<td>Number of arguments (@1) exceeds the maximum (@2) number of EXCEPTION USING arguments</td>
</tr>
<tr>
<td>-901</td>
<td>336397331</td>
<td>dsq_string_byte_length</td>
<td>String literal with @1 bytes exceeds the maximum length of @2 bytes</td>
</tr>
<tr>
<td>-901</td>
<td>336397332</td>
<td>dsq_string_char_length</td>
<td>String literal with @1 characters exceeds the maximum length of @2 characters for the @3 character set</td>
</tr>
<tr>
<td>-901</td>
<td>336397333</td>
<td>dsq_max_nesting</td>
<td>Too many BEGIN...END nesting. Maximum level is @1</td>
</tr>
<tr>
<td>-902</td>
<td>335544333</td>
<td>bug_check</td>
<td>internal Firebird consistency check (@1)</td>
</tr>
<tr>
<td>-902</td>
<td>335544335</td>
<td>db_corrupt</td>
<td>database file appears corrupt (@1)</td>
</tr>
<tr>
<td>-902</td>
<td>335544344</td>
<td>io_error</td>
<td>I/O error during &quot;@1&quot; operation for file &quot;@2&quot;</td>
</tr>
<tr>
<td>-902</td>
<td>335544346</td>
<td>metadata_corrupt</td>
<td>corrupt system table</td>
</tr>
<tr>
<td>-902</td>
<td>335544373</td>
<td>sys_request</td>
<td>operating system directive @1 failed</td>
</tr>
<tr>
<td>-902</td>
<td>335544384</td>
<td>badblk</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544385</td>
<td>invpoolcl</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544387</td>
<td>relbadblk</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544388</td>
<td>blktoobig</td>
<td>block size exceeds implementation restriction</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>----------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>-902</td>
<td>335544394</td>
<td>badodsver</td>
<td>incompatible version of on-disk structure</td>
</tr>
<tr>
<td>-902</td>
<td>335544397</td>
<td>dirtpage</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544398</td>
<td>waifortra</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544399</td>
<td>doubleloc</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544400</td>
<td>nodnotfnd</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544401</td>
<td>dupnodfnd</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544402</td>
<td>locnotmar</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544404</td>
<td>corrupt</td>
<td>database corrupted</td>
</tr>
<tr>
<td>-902</td>
<td>335544405</td>
<td>badpage</td>
<td>checksum error on database page @1</td>
</tr>
<tr>
<td>-902</td>
<td>335544406</td>
<td>badindex</td>
<td>index is broken</td>
</tr>
<tr>
<td>-902</td>
<td>335544409</td>
<td>trareqmis</td>
<td>transaction—request mismatch (synchronization error)</td>
</tr>
<tr>
<td>-902</td>
<td>335544410</td>
<td>badhndcnt</td>
<td>bad handle count</td>
</tr>
<tr>
<td>-902</td>
<td>335544411</td>
<td>wrotpbver</td>
<td>wrong version of transaction parameter block</td>
</tr>
<tr>
<td>-902</td>
<td>335544412</td>
<td>wroblrver</td>
<td>unsupported BLR version (expected @1, encountered @2)</td>
</tr>
<tr>
<td>-902</td>
<td>335544413</td>
<td>wrodpbver</td>
<td>wrong version of database parameter block</td>
</tr>
<tr>
<td>-902</td>
<td>335544415</td>
<td>badrelation</td>
<td>database corrupted</td>
</tr>
<tr>
<td>-902</td>
<td>335544416</td>
<td>nodetach</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544417</td>
<td>notremote</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544422</td>
<td>dbfile</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544423</td>
<td>orphan</td>
<td>internal error</td>
</tr>
<tr>
<td>-902</td>
<td>335544432</td>
<td>lockmanerr</td>
<td>lock manager error</td>
</tr>
<tr>
<td>-902</td>
<td>335544436</td>
<td>sqllerr</td>
<td>SQL error code = @1</td>
</tr>
<tr>
<td>-902</td>
<td>335544448</td>
<td>bad_sec_info</td>
<td></td>
</tr>
<tr>
<td>-902</td>
<td>335544449</td>
<td>invalid_sec_info</td>
<td></td>
</tr>
<tr>
<td>-902</td>
<td>335544470</td>
<td>buf_invalid</td>
<td>cache buffer for page @1 invalid</td>
</tr>
<tr>
<td>-902</td>
<td>335544471</td>
<td>indexnotdefined</td>
<td>there is no index in table @1 with id @2</td>
</tr>
<tr>
<td>-902</td>
<td>335544472</td>
<td>login</td>
<td>Your user name and password are not defined. Ask your database administrator to set up a Firebird login.</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-902</td>
<td>335544478</td>
<td>jrn_enable</td>
<td>enable journal for database before starting online dump</td>
</tr>
<tr>
<td>-902</td>
<td>335544479</td>
<td>old_failure</td>
<td>online dump failure. Retry dump</td>
</tr>
<tr>
<td>-902</td>
<td>335544480</td>
<td>old_in_progress</td>
<td>an online dump is already in progress</td>
</tr>
<tr>
<td>-902</td>
<td>335544481</td>
<td>old_no_space</td>
<td>no more disk/tape space. Cannot continue online dump</td>
</tr>
<tr>
<td>-902</td>
<td>335544482</td>
<td>no_wal_no_jrn</td>
<td>journaling allowed only if database has Write-ahead Log</td>
</tr>
<tr>
<td>-902</td>
<td>335544483</td>
<td>num_old_files</td>
<td>maximum number of online dump files that can be specified is 16</td>
</tr>
<tr>
<td>-902</td>
<td>335544484</td>
<td>wal_file_open</td>
<td>error in opening Write-ahead Log file during recovery</td>
</tr>
<tr>
<td>-902</td>
<td>335544486</td>
<td>wal_failure</td>
<td>Write-ahead log subsystem failure</td>
</tr>
<tr>
<td>-902</td>
<td>335544505</td>
<td>no_archive</td>
<td>must specify archive file when enabling long term journal for databases with round-robin log files</td>
</tr>
<tr>
<td>-902</td>
<td>335544506</td>
<td>shutinprog</td>
<td>database @1 shutdown in progress</td>
</tr>
<tr>
<td>-902</td>
<td>335544520</td>
<td>jrn_present</td>
<td>long-term journaling already enabled</td>
</tr>
<tr>
<td>-902</td>
<td>335544528</td>
<td>shutdown</td>
<td>database @1 shutdown</td>
</tr>
<tr>
<td>-902</td>
<td>335544557</td>
<td>shutfail</td>
<td>database shutdown unsuccessful</td>
</tr>
<tr>
<td>-902</td>
<td>335544564</td>
<td>no_jrn</td>
<td>long-term journaling not enabled</td>
</tr>
<tr>
<td>-902</td>
<td>335544569</td>
<td>dsql_error</td>
<td>Dynamic SQL Error</td>
</tr>
<tr>
<td>-902</td>
<td>335544653</td>
<td>psw_attach</td>
<td>cannot attach to password database</td>
</tr>
<tr>
<td>-902</td>
<td>335544654</td>
<td>psw_start_trans</td>
<td>cannot start transaction for password database</td>
</tr>
<tr>
<td>-902</td>
<td>335544717</td>
<td>err_stack_limit</td>
<td>stack size insufficient to execute current request</td>
</tr>
<tr>
<td>-902</td>
<td>335544721</td>
<td>network_error</td>
<td>Unable to complete network request to host &quot;@1&quot;.</td>
</tr>
<tr>
<td>-902</td>
<td>335544722</td>
<td>net_connect_err</td>
<td>Failed to establish a connection.</td>
</tr>
<tr>
<td>-902</td>
<td>335544723</td>
<td>net_connect_listen_err</td>
<td>Error while listening for an incoming connection.</td>
</tr>
<tr>
<td>-902</td>
<td>335544724</td>
<td>net_event_connect_err</td>
<td>Failed to establish a secondary connection for event processing.</td>
</tr>
<tr>
<td>-902</td>
<td>335544725</td>
<td>net_event_listen_err</td>
<td>Error while listening for an incoming event connection request.</td>
</tr>
</tbody>
</table>
## Appendix B: Exception Codes and Messages

<table>
<thead>
<tr>
<th>GDSCODE</th>
<th>Symbol</th>
<th>Message Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>-902 335544726</td>
<td>net_read_err</td>
<td>Error reading data from the connection.</td>
</tr>
<tr>
<td>-902 335544727</td>
<td>net_write_err</td>
<td>Error writing data to the connection.</td>
</tr>
<tr>
<td>-902 335544732</td>
<td>unsupported_network_drive</td>
<td>Access to databases on file servers is not supported.</td>
</tr>
<tr>
<td>-902 335544733</td>
<td>io_create_err</td>
<td>Error while trying to create file</td>
</tr>
<tr>
<td>-902 335544734</td>
<td>io_open_err</td>
<td>Error while trying to open file</td>
</tr>
<tr>
<td>-902 335544735</td>
<td>io_close_err</td>
<td>Error while trying to close file</td>
</tr>
<tr>
<td>-902 335544736</td>
<td>io_read_err</td>
<td>Error while trying to read from file</td>
</tr>
<tr>
<td>-902 335544737</td>
<td>io_write_err</td>
<td>Error while trying to write to file</td>
</tr>
<tr>
<td>-902 335544738</td>
<td>io_delete_err</td>
<td>Error while trying to delete file</td>
</tr>
<tr>
<td>-902 335544739</td>
<td>io_access_err</td>
<td>Error while trying to access file</td>
</tr>
<tr>
<td>-902 335544745</td>
<td>login_same_as_role_name</td>
<td>Your login @1 is same as one of the SQL role name. Ask your database administrator to set up a valid Firebird login.</td>
</tr>
<tr>
<td>-902 335544791</td>
<td>file_in_use</td>
<td>The file @1 is currently in use by another process. Try again later.</td>
</tr>
<tr>
<td>-902 335544795</td>
<td>unexp_spb_form</td>
<td>unexpected item in service parameter block, expected @1</td>
</tr>
<tr>
<td>-902 335544809</td>
<td>extern_func_dir_error</td>
<td>Function @1 is in @2, which is not in a permitted directory for external functions.</td>
</tr>
<tr>
<td>-902 335544819</td>
<td>io_32bit_exceeded_err</td>
<td>File exceeded maximum size of 2GB. Add another database file or use a 64 bit I/O version of Firebird.</td>
</tr>
<tr>
<td>-902 335544831</td>
<td>conf_access_denied</td>
<td>Use of @1 at location @2 is not allowed by server configuration</td>
</tr>
<tr>
<td>-902 335544834</td>
<td>cursor_not_open</td>
<td>Cursor is not open</td>
</tr>
<tr>
<td>-902 335544841</td>
<td>cursor_already_open</td>
<td>Cursor is already open</td>
</tr>
<tr>
<td>-902 335544856</td>
<td>att_shutdown</td>
<td>connection shutdown</td>
</tr>
<tr>
<td>-902 335544882</td>
<td>long_login</td>
<td>Login name too long (@1 characters, maximum allowed @2)</td>
</tr>
<tr>
<td>-902 335544936</td>
<td>psw_db_error</td>
<td>Security database error</td>
</tr>
<tr>
<td>-902 335544970</td>
<td>missing_required_spb</td>
<td>Missing required item @1 in service parameter block</td>
</tr>
<tr>
<td>-902 335544971</td>
<td>net_server_shutdown</td>
<td>@1 server is shutdown</td>
</tr>
<tr>
<td>SQLCODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>-902</td>
<td>335544974</td>
<td>no_threads</td>
</tr>
<tr>
<td>-902</td>
<td>335544975</td>
<td>net_event_connect_timeout</td>
</tr>
<tr>
<td>-902</td>
<td>335544984</td>
<td>instance_conflict</td>
</tr>
<tr>
<td>-902</td>
<td>335544987</td>
<td>no_trusted_spb</td>
</tr>
<tr>
<td>-902</td>
<td>335545029</td>
<td>missing_data_structures</td>
</tr>
<tr>
<td>-902</td>
<td>335545030</td>
<td>protect_sys_tab</td>
</tr>
<tr>
<td>-902</td>
<td>335545032</td>
<td>wroblrver2</td>
</tr>
<tr>
<td>-902</td>
<td>335545043</td>
<td>decrypt_error</td>
</tr>
<tr>
<td>-902</td>
<td>335545044</td>
<td>no_providers</td>
</tr>
<tr>
<td>-902</td>
<td>335545053</td>
<td>miss_config</td>
</tr>
<tr>
<td>-902</td>
<td>335545054</td>
<td>conf_line</td>
</tr>
<tr>
<td>-902</td>
<td>335545055</td>
<td>conf_include</td>
</tr>
<tr>
<td>-902</td>
<td>335545056</td>
<td>include_depth</td>
</tr>
<tr>
<td>-902</td>
<td>335545057</td>
<td>include_miss</td>
</tr>
<tr>
<td>-902</td>
<td>335545060</td>
<td>sec_context</td>
</tr>
<tr>
<td>-902</td>
<td>335545061</td>
<td>multi_segment</td>
</tr>
<tr>
<td>-902</td>
<td>335545062</td>
<td>login_changed</td>
</tr>
<tr>
<td>-902</td>
<td>335545063</td>
<td>auth_handshake_limit</td>
</tr>
<tr>
<td>-902</td>
<td>335545064</td>
<td>wirecrypt_incompatible</td>
</tr>
<tr>
<td>-902</td>
<td>335545065</td>
<td>miss_wirecrypt</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>-902</td>
<td>335545066</td>
<td>wirecrypt_key</td>
</tr>
<tr>
<td>-902</td>
<td>335545067</td>
<td>wirecrypt_plugin</td>
</tr>
<tr>
<td>-902</td>
<td>335545068</td>
<td>secdb_name</td>
</tr>
<tr>
<td>-902</td>
<td>335545069</td>
<td>auth_data</td>
</tr>
<tr>
<td>-902</td>
<td>335545070</td>
<td>auth_datalength</td>
</tr>
<tr>
<td>-902</td>
<td>335545106</td>
<td>login_error</td>
</tr>
<tr>
<td>-902</td>
<td>335545107</td>
<td>already_opened</td>
</tr>
<tr>
<td>-902</td>
<td>335545108</td>
<td>bad_crypt_key</td>
</tr>
<tr>
<td>-904</td>
<td>335544324</td>
<td>bad_db_handle</td>
</tr>
<tr>
<td>-904</td>
<td>335544375</td>
<td>unavailable</td>
</tr>
<tr>
<td>-904</td>
<td>335544381</td>
<td>imp_exc</td>
</tr>
<tr>
<td>-904</td>
<td>335544386</td>
<td>nopoolids</td>
</tr>
<tr>
<td>-904</td>
<td>335544389</td>
<td>bufexh</td>
</tr>
<tr>
<td>-904</td>
<td>335544391</td>
<td>bufinuse</td>
</tr>
<tr>
<td>-904</td>
<td>335544393</td>
<td>reqinuse</td>
</tr>
<tr>
<td>-904</td>
<td>335544424</td>
<td>no_lock_mg</td>
</tr>
<tr>
<td>-904</td>
<td>335544430</td>
<td>virmemexh</td>
</tr>
<tr>
<td>-904</td>
<td>335544451</td>
<td>update_conflict</td>
</tr>
<tr>
<td>-904</td>
<td>335544453</td>
<td>obj_in_use</td>
</tr>
<tr>
<td>-904</td>
<td>335544455</td>
<td>shadow_accessed</td>
</tr>
<tr>
<td>-904</td>
<td>335544460</td>
<td>shadow_missing</td>
</tr>
<tr>
<td>-904</td>
<td>335544661</td>
<td>index_root_page_full</td>
</tr>
<tr>
<td>-904</td>
<td>335544676</td>
<td>sort_mem_err</td>
</tr>
<tr>
<td>SQL- CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>-904</td>
<td>335544683</td>
<td>req_depth_exceeded</td>
</tr>
<tr>
<td>-904</td>
<td>335544758</td>
<td>sort_rec_size_err</td>
</tr>
<tr>
<td>-904</td>
<td>335544761</td>
<td>too_many_handles</td>
</tr>
<tr>
<td>-904</td>
<td>335544762</td>
<td>optimizer_blk_exc</td>
</tr>
<tr>
<td>-904</td>
<td>335544792</td>
<td>service_att_err</td>
</tr>
<tr>
<td>-904</td>
<td>335544799</td>
<td>svc_name_missing</td>
</tr>
<tr>
<td>-904</td>
<td>335544813</td>
<td>optimizer_between_err</td>
</tr>
<tr>
<td>-904</td>
<td>335544827</td>
<td>exec_sql_invalid_arg</td>
</tr>
<tr>
<td>-904</td>
<td>335544828</td>
<td>exec_sql_invalid_req</td>
</tr>
<tr>
<td>-904</td>
<td>335544829</td>
<td>exec_sql_invalid_var</td>
</tr>
<tr>
<td>-904</td>
<td>335544830</td>
<td>exec_sql_max_call_exceeded</td>
</tr>
<tr>
<td>-904</td>
<td>335544832</td>
<td>wrong_backup_state</td>
</tr>
<tr>
<td>-904</td>
<td>335544833</td>
<td>wal_backup_err</td>
</tr>
<tr>
<td>-904</td>
<td>335544852</td>
<td>partner_idx_incompat_type</td>
</tr>
<tr>
<td>-904</td>
<td>335544857</td>
<td>blobtoobig</td>
</tr>
<tr>
<td>-904</td>
<td>335544862</td>
<td>record_lock_not_supp</td>
</tr>
<tr>
<td>-904</td>
<td>335544863</td>
<td>partner_idx_not_found</td>
</tr>
<tr>
<td>-904</td>
<td>335544864</td>
<td>tra_num_exc</td>
</tr>
<tr>
<td>-904</td>
<td>335544865</td>
<td>field_disappeared</td>
</tr>
<tr>
<td>-904</td>
<td>335544878</td>
<td>concurrent_transaction</td>
</tr>
<tr>
<td>-904</td>
<td>335544935</td>
<td>circular_computed</td>
</tr>
<tr>
<td>-904</td>
<td>335544992</td>
<td>lock_dir_access</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>-904</td>
<td>335545020</td>
<td>request_outdated</td>
</tr>
<tr>
<td>-904</td>
<td>335545096</td>
<td>read_conflict</td>
</tr>
<tr>
<td>-906</td>
<td>335544452</td>
<td>unlicensed</td>
</tr>
<tr>
<td>-906</td>
<td>335544744</td>
<td>max_att_exceeded</td>
</tr>
<tr>
<td>-909</td>
<td>335544667</td>
<td>drdb_completed_with_errs</td>
</tr>
<tr>
<td>-911</td>
<td>335544459</td>
<td>rec_in_limbo</td>
</tr>
<tr>
<td>-913</td>
<td>335544336</td>
<td>deadlock</td>
</tr>
<tr>
<td>-922</td>
<td>335544323</td>
<td>bad_db_format</td>
</tr>
<tr>
<td>-923</td>
<td>335544421</td>
<td>connect_reject</td>
</tr>
<tr>
<td>-923</td>
<td>335544461</td>
<td>cant_validate</td>
</tr>
<tr>
<td>-923</td>
<td>335544462</td>
<td>cant_start_journal</td>
</tr>
<tr>
<td>-923</td>
<td>335544464</td>
<td>cant_start_logging</td>
</tr>
<tr>
<td>-924</td>
<td>335544325</td>
<td>bad_dp ContentType</td>
</tr>
<tr>
<td>-924</td>
<td>335544433</td>
<td>journerr</td>
</tr>
<tr>
<td>-924</td>
<td>335544441</td>
<td>bad_detach</td>
</tr>
<tr>
<td>-924</td>
<td>335544648</td>
<td>conn_lost</td>
</tr>
<tr>
<td>-924</td>
<td>335544972</td>
<td>bad_conn_str</td>
</tr>
<tr>
<td>-924</td>
<td>335545085</td>
<td>baddpb_damaged_mode</td>
</tr>
<tr>
<td>-924</td>
<td>335545086</td>
<td>baddpb_buffers_range</td>
</tr>
<tr>
<td>-924</td>
<td>335545087</td>
<td>baddpb_temp_buffers</td>
</tr>
<tr>
<td>-926</td>
<td>335544447</td>
<td>no rollback</td>
</tr>
<tr>
<td>-999</td>
<td>335544689</td>
<td>ib_error</td>
</tr>
</tbody>
</table>
Appendix C: Reserved Words and Keywords

Reserved words are part of the Firebird SQL language. They cannot be used as identifiers (e.g. as table or procedure names), except when enclosed in double quotes in Dialect 3. However, you should avoid this unless you have a compelling reason.

Keywords are also part of the language. They have a special meaning when used in the proper context, but they are not reserved for Firebird's own and exclusive use. You can use them as identifiers without double-quoting.

### Reserved words

Full list of reserved words in Firebird 3.0:

<table>
<thead>
<tr>
<th>Reserved word</th>
<th>Reserved word</th>
<th>Reserved word</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>ADMIN</td>
<td>ALL</td>
</tr>
<tr>
<td>ALTER</td>
<td>AND</td>
<td>ANY</td>
</tr>
<tr>
<td>AS</td>
<td>AT</td>
<td>AVG</td>
</tr>
<tr>
<td>BEGIN</td>
<td>BETWEEN</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BIT_LENGTH</td>
<td>BLOB</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>BOTH</td>
<td>BY</td>
<td>CASE</td>
</tr>
<tr>
<td>CAST</td>
<td>CHAR</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>CHARACTER_LENGTH</td>
<td>CHAR_LENGTH</td>
<td>CHECK</td>
</tr>
<tr>
<td>CLOSE</td>
<td>COLLATE</td>
<td>COLUMN</td>
</tr>
<tr>
<td>COMMIT</td>
<td>CONNECT</td>
<td>CONSTRAINT</td>
</tr>
<tr>
<td>CORR</td>
<td>COUNT</td>
<td>COVAR_POP</td>
</tr>
<tr>
<td>COVAR_SAMP</td>
<td>CREATE</td>
<td>CROSS</td>
</tr>
<tr>
<td>CURRENT</td>
<td>CURRENT_CONNECTION</td>
<td>CURRENT_DATE</td>
</tr>
<tr>
<td>CURRENT_ROLE</td>
<td>CURRENT_TIME</td>
<td>CURRENT_TIMESTAMP</td>
</tr>
<tr>
<td>CURRENT_TRANSACTION</td>
<td>CURRENT_USER</td>
<td>CURSOR</td>
</tr>
<tr>
<td>DATE</td>
<td>DAY</td>
<td>DEC</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECLARE</td>
<td>DEFAULT</td>
</tr>
<tr>
<td>DELETE</td>
<td>DELETING</td>
<td>DETERMINISTIC</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>DISTINCT</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>DROP</td>
<td>ELSE</td>
<td>END</td>
</tr>
<tr>
<td>ESCAPE</td>
<td>EXECUTE</td>
<td>EXISTS</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>EXTRACT</td>
<td>FALSE</td>
</tr>
<tr>
<td>FETCH</td>
<td>FILTER</td>
<td>FLOAT</td>
</tr>
<tr>
<td>FOR</td>
<td>FOREIGN</td>
<td>FROM</td>
</tr>
<tr>
<td>FULL</td>
<td>FUNCTION</td>
<td>GDSCODE</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>GRANT</td>
<td>GROUP</td>
</tr>
<tr>
<td>HAVING</td>
<td>HOUR</td>
<td>IN</td>
</tr>
<tr>
<td>INDEX</td>
<td>INNER</td>
<td>INSENSITIVE</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>INSERT</td>
<td>INSERTING</td>
<td>INT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTO</td>
<td>IS</td>
</tr>
<tr>
<td>JOIN</td>
<td>LEADING</td>
<td>LEFT</td>
</tr>
<tr>
<td>LIKE</td>
<td>LONG</td>
<td>LOWER</td>
</tr>
<tr>
<td>MAX</td>
<td>MERGE</td>
<td>MIN</td>
</tr>
<tr>
<td>MINUTE</td>
<td>MONTH</td>
<td>NATIONAL</td>
</tr>
<tr>
<td>NATURAL</td>
<td>NCHAR</td>
<td>NO</td>
</tr>
<tr>
<td>NOT</td>
<td>NULL</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>OCTET_LENGTH</td>
<td>OF</td>
<td>OFFSET</td>
</tr>
<tr>
<td>ON</td>
<td>ONLY</td>
<td>OPEN</td>
</tr>
<tr>
<td>OR</td>
<td>ORDER</td>
<td>OUTER</td>
</tr>
<tr>
<td>OVER</td>
<td>PARAMETER</td>
<td>PLAN</td>
</tr>
<tr>
<td>POSITION</td>
<td>POST_EVENT</td>
<td>PRECISION</td>
</tr>
<tr>
<td>PRIMARY</td>
<td>PROCEDURE</td>
<td>RDB$DB_KEY</td>
</tr>
<tr>
<td>RDB$RECORD_VERSION</td>
<td>REAL</td>
<td>RECORD_VERSION</td>
</tr>
<tr>
<td>RECREATE</td>
<td>RECURSIVE</td>
<td>REFERENCES</td>
</tr>
<tr>
<td>REGR_AVGX</td>
<td>REGR_AVGY</td>
<td>REGR_COUNT</td>
</tr>
<tr>
<td>REGR_INTERCEPT</td>
<td>REGR_R2</td>
<td>REGR_SLOPE</td>
</tr>
<tr>
<td>REGR_SXX</td>
<td>REGR_SXY</td>
<td>REGR_SYY</td>
</tr>
<tr>
<td>RELEASE</td>
<td>RETURN</td>
<td>RETURNING_VALUES</td>
</tr>
<tr>
<td>RETURNS</td>
<td>REVOKE</td>
<td>RIGHT</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>ROW</td>
<td>ROWS</td>
</tr>
<tr>
<td>ROW_COUNT</td>
<td>SAVEPOINT</td>
<td>SCROLL</td>
</tr>
<tr>
<td>SECOND</td>
<td>SELECT</td>
<td>SENSITIVE</td>
</tr>
<tr>
<td>SET</td>
<td>SIMILAR</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>SOME</td>
<td>SQLCODE</td>
<td>SQLSTATE</td>
</tr>
<tr>
<td>START</td>
<td>STDDEV_POP</td>
<td>STDDEV_SAMP</td>
</tr>
<tr>
<td>SUM</td>
<td>TABLE</td>
<td>THEN</td>
</tr>
<tr>
<td>TIME</td>
<td>TIMESTAMP</td>
<td>TO</td>
</tr>
<tr>
<td>TRAILING</td>
<td>TRIGGER</td>
<td>TRIM</td>
</tr>
<tr>
<td>TRUE</td>
<td>UNION</td>
<td>UNIQUE</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>UPDATE</td>
<td>UPDATING</td>
</tr>
<tr>
<td>UPPER</td>
<td>USER</td>
<td>USING</td>
</tr>
<tr>
<td>VALUE</td>
<td>VALUES</td>
<td>VARCHART</td>
</tr>
<tr>
<td>VARIABLE</td>
<td>VARYING</td>
<td>VAR_POP</td>
</tr>
<tr>
<td>VAR_SAMP</td>
<td>VIEW</td>
<td>WHEN</td>
</tr>
<tr>
<td>WHERE</td>
<td>WHILE</td>
<td>WITH</td>
</tr>
<tr>
<td>YEAR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Keywords

The following terms have a special meaning in Firebird 3.0 DSQL. Some of them are also reserved words, others are not.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>ABSOLUTE</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>ACCENT</td>
</tr>
<tr>
<td><code>^</code></td>
<td>ACOS</td>
</tr>
<tr>
<td><code>^=</code></td>
<td>ACTION</td>
</tr>
<tr>
<td><code>^&gt;</code></td>
<td>ADD</td>
</tr>
<tr>
<td><code>:=</code></td>
<td>ADMIN</td>
</tr>
<tr>
<td><code>!&gt;</code></td>
<td>ALL</td>
</tr>
<tr>
<td>`!</td>
<td>=`</td>
</tr>
<tr>
<td>`</td>
<td>=`</td>
</tr>
<tr>
<td>`</td>
<td>=`</td>
</tr>
<tr>
<td>`</td>
<td>=`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
</tr>
</tbody>
</table>

---

### Appendix C: Reserved Words and Keywords

603
Appendix C: Reserved Words and Keywords

CURRENT  CURRENT CONNECTION  CURRENT DATE
CURRENT ROLE  CURRENT TIME  CURRENT DATE
CURRENT TRANSACTION  CURRENT USER  CURRENT_TIMESTAMP
DATA  DATABASE  DATE
DATEADD  DATEDIFF  DAY
DB_KEY  DDL  DEC
DECIMAL  DECLARE  DECODE
DECRYPT  DEFAULT  DELETE
DELETING  DENSE_RANK  DESC
DESCENDING  DESCRIPTOR  DETERMINISTIC
DIFFERENCE  DISCONNECT  DISTINCT
DO  DOMAIN  DOUBLE
DROP  ELSE  ENCRYPT
END  ENGINE  ENTRY_POINT
ESCAPE  EXCEPTION  EXECUTE
EXISTS  EXIT  EXP
EXTERNAL  EXTRACT  FALSE
FETCH  FILE  FILTER
FIRST  FIRSTNAME  FIRST_VALUE
FLOAT  FLOOR  FOR
FOREIGN  FREE_IT  FROM
FULL  FUNCTION  GDSCODE
GENERATED  GENERATOR  GEN_ID
GEN_UUID  GLOBAL  GRANT
GRANTED  GROUP  HASH
HAVING  HOUR  IDENTITY
IF  IGNORE  IIF
IN  INACTIVE  INCREMENT
INDEX  INNER  INPUT_TYPE
INSENSITIVE  INSERT  INSERTING
INT  INTEGER  INTO
IS  ISOLATION  JOIN
KEY  LAG  LAST
LASTNAME  LAST_VALUE  LEAD
LEADING  LEAVE  LEFT
LENGTH  LEVEL  LIKE
LIMBO  LINGER  LIST
LN  LOCALTIME  LOCALTIMESTAMP
LOCK  LOG  LOG10
<table>
<thead>
<tr>
<th>Long</th>
<th>Lower</th>
<th>Lpad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>Mapping</td>
<td>Matched</td>
</tr>
<tr>
<td>Matching</td>
<td>Max</td>
<td>Maxvalue</td>
</tr>
<tr>
<td>Merge</td>
<td>Middlename</td>
<td>Millisecond</td>
</tr>
<tr>
<td>Min</td>
<td>Minute</td>
<td>Minvalue</td>
</tr>
<tr>
<td>Mod</td>
<td>Module_Name</td>
<td>Month</td>
</tr>
<tr>
<td>Name</td>
<td>Names</td>
<td>National</td>
</tr>
<tr>
<td>Natural</td>
<td>Nchar</td>
<td>Next</td>
</tr>
<tr>
<td>No</td>
<td>Not</td>
<td>Nth_value</td>
</tr>
<tr>
<td>Null</td>
<td>Nullif</td>
<td>Nulls</td>
</tr>
<tr>
<td>Numeric</td>
<td>Octet_length</td>
<td>Of</td>
</tr>
<tr>
<td>Offset</td>
<td>On</td>
<td>Only</td>
</tr>
<tr>
<td>Open</td>
<td>Option</td>
<td>Or</td>
</tr>
<tr>
<td>Order</td>
<td>Os_Name</td>
<td>Outer</td>
</tr>
<tr>
<td>Output_Type</td>
<td>Over</td>
<td>Overflow</td>
</tr>
<tr>
<td>Overlay</td>
<td>Package</td>
<td>Pad</td>
</tr>
<tr>
<td>Page</td>
<td>Pages</td>
<td>Page_size</td>
</tr>
<tr>
<td>Parameter</td>
<td>Partition</td>
<td>Password</td>
</tr>
<tr>
<td>Pi</td>
<td>Placing</td>
<td>Plan</td>
</tr>
<tr>
<td>Plugin</td>
<td>Position</td>
<td>Post_event</td>
</tr>
<tr>
<td>Power</td>
<td>Precision</td>
<td>Preserve</td>
</tr>
<tr>
<td>Primary</td>
<td>Prior</td>
<td>Privileges</td>
</tr>
<tr>
<td>Procedure</td>
<td>Protected</td>
<td>Rand</td>
</tr>
<tr>
<td>Rank</td>
<td>Rdb$db_key</td>
<td>Rdb$Get_Context</td>
</tr>
<tr>
<td>Rdb$Record_Version</td>
<td>Rdb$Set_Context</td>
<td>Rdb_Get_Context</td>
</tr>
<tr>
<td>Rdb_Set_Context</td>
<td>Read</td>
<td>Real</td>
</tr>
<tr>
<td>Record_Version</td>
<td>Recreate</td>
<td>Recursive</td>
</tr>
<tr>
<td>References</td>
<td>Regr_avgx</td>
<td>Regr_avgy</td>
</tr>
<tr>
<td>Regr_Count</td>
<td>Regr_intercept</td>
<td>Regr_r2</td>
</tr>
<tr>
<td>Regr_Slope</td>
<td>Regr_sxx</td>
<td>Regr_sxy</td>
</tr>
<tr>
<td>Regr_Syy</td>
<td>Relative</td>
<td>Release</td>
</tr>
<tr>
<td>Replace</td>
<td>Requests</td>
<td>Reserv</td>
</tr>
<tr>
<td>Reserving</td>
<td>Restart</td>
<td>Restrict</td>
</tr>
<tr>
<td>Retain</td>
<td>Return</td>
<td>Returning</td>
</tr>
<tr>
<td>Returning_Values</td>
<td>Returns</td>
<td>Reverse</td>
</tr>
<tr>
<td>Revoke</td>
<td>Right</td>
<td>Role</td>
</tr>
<tr>
<td>Rollback</td>
<td>Round</td>
<td>Returning</td>
</tr>
<tr>
<td>Rows</td>
<td>Row_count</td>
<td>Row_number</td>
</tr>
<tr>
<td>Rpad</td>
<td>Savepoint</td>
<td>Scalar_array</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>SCROLL</td>
<td>SECOND</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>----------</td>
</tr>
<tr>
<td>SEGMENT</td>
<td>SELECT</td>
<td>SENSITIVE</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>SERVERWIDE</td>
<td>SET</td>
</tr>
<tr>
<td>SHADOW</td>
<td>SHARED</td>
<td>SIGN</td>
</tr>
<tr>
<td>SIMILAR</td>
<td>SIN</td>
<td>SINGULAR</td>
</tr>
<tr>
<td>SINH</td>
<td>SIZE</td>
<td>SKIP</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SNAPSHOT</td>
<td>SOME</td>
</tr>
<tr>
<td>SORT</td>
<td>SOURCE</td>
<td>SPACE</td>
</tr>
<tr>
<td>SQLCODE</td>
<td>SQLSTATE</td>
<td>SQRT</td>
</tr>
<tr>
<td>STABILITY</td>
<td>START</td>
<td>STARTING</td>
</tr>
<tr>
<td>STARTS</td>
<td>STATEMENT</td>
<td>STATISTICS</td>
</tr>
<tr>
<td>STDDEV_POP</td>
<td>STDDEV_SAMP</td>
<td>SUBSTRING</td>
</tr>
<tr>
<td>SUB_TYPE</td>
<td>SUM</td>
<td>SUSPEND</td>
</tr>
<tr>
<td>TABLE</td>
<td>TAGS</td>
<td>TAN</td>
</tr>
<tr>
<td>TANH</td>
<td>TEMPORARY</td>
<td>THEN</td>
</tr>
<tr>
<td>TIME</td>
<td>TIMEOUT</td>
<td>TIMESTMP</td>
</tr>
<tr>
<td>TO</td>
<td>TRAILING</td>
<td>TRANSACTION</td>
</tr>
<tr>
<td>TRIGGER</td>
<td>TRIM</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUNC</td>
<td>TRUSTED</td>
<td>TWO_PHASE</td>
</tr>
<tr>
<td>TYPE</td>
<td>UNCOMMITTED</td>
<td>UNDO</td>
</tr>
<tr>
<td>UNION</td>
<td>UNIQUE</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>UPDATE</td>
<td>UPDATING</td>
<td>UPPER</td>
</tr>
<tr>
<td>USAGE</td>
<td>USER</td>
<td>USING</td>
</tr>
<tr>
<td>UUID_TO_CHAR</td>
<td>VALUE</td>
<td>VALUES</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARIABLE</td>
<td>VARYING</td>
</tr>
<tr>
<td>VAR_POP</td>
<td>VAR_SAMP</td>
<td>VIEW</td>
</tr>
<tr>
<td>WAIT</td>
<td>WEEK</td>
<td>WEEKDAY</td>
</tr>
<tr>
<td>WHEN</td>
<td>WHERE</td>
<td>WHILE</td>
</tr>
<tr>
<td>WITH</td>
<td>WORK</td>
<td>WRITE</td>
</tr>
<tr>
<td>YEAR</td>
<td>YEARDAY</td>
<td></td>
</tr>
</tbody>
</table>
Appendix D: System Tables

When you create a database, the Firebird engine creates a lot of system tables. Metadata — the descriptions and attributes of all database objects — are stored in these system tables.

System table identifiers all begin with the prefix RDB$.

List of System Tables

**RDB$AUTH_MAPPING**
Stores authentication and other security mappings

**RDB$BACKUP_HISTORY**
History of backups performed using nBackup

**RDB$CHARACTER_SETS**
Names and describes the character sets available in the database

**RDB$CHECK_CONSTRAINTS**
Cross references between the names of constraints (NOT NULL constraints, CHECK constraints and ON UPDATE and ON DELETE clauses in foreign key constraints) and their associated system-generated triggers

**RDB$COLLATIONS**
Collation sequences for all character sets

**RDB$DATABASE**
Basic information about the database

**RDB$DB_CREATORS**
A list of users granted the CREATE DATABASE privilege when using the specified database as a security database

**RDB$DEPENDENCIES**
Information about dependencies between database objects

**RDB$EXCEPTIONS**
Custom database exceptions

**RDB$FIELDS**
Column and domain definitions, both system and custom

**RDB$FIELD_DIMENSIONS**
Dimensions of array columns

**RDB$FILES**
Information about secondary files and shadow files
<table>
<thead>
<tr>
<th>Table Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$FILTERS</td>
<td>Information about BLOB filters</td>
</tr>
<tr>
<td>RDB$FORMATS</td>
<td>Information about changes in the formats of tables</td>
</tr>
<tr>
<td>RDB$FUNCTIONS</td>
<td>Information about external functions</td>
</tr>
<tr>
<td>RDB$FUNCTION_ARGUMENTS</td>
<td>Attributes of the parameters of external functions</td>
</tr>
<tr>
<td>RDB$GENERATORS</td>
<td>Information about generators (sequences)</td>
</tr>
<tr>
<td>RDB$INDEX_SEGMENTS</td>
<td>Segments and index positions</td>
</tr>
<tr>
<td>RDB$INDICES</td>
<td>Definitions of all indexes in the database (system- or user-defined)</td>
</tr>
<tr>
<td>RDB$LOG_FILES</td>
<td>Not used in the current version</td>
</tr>
<tr>
<td>RDB$PACKAGES</td>
<td>Stores the definition (header and body) of SQL packages</td>
</tr>
<tr>
<td>RDB$PAGES</td>
<td>Information about database pages</td>
</tr>
<tr>
<td>RDB$PROcedures</td>
<td>Definitions of stored procedures</td>
</tr>
<tr>
<td>RDB$PROCEDURE_PARAMETERS</td>
<td>Parameters of stored procedures</td>
</tr>
<tr>
<td>RDB$REF_CONSTRAINTS</td>
<td>Definitions of referential constraints (foreign keys)</td>
</tr>
<tr>
<td>RDB$RELATIONS</td>
<td>Headers of tables and views</td>
</tr>
<tr>
<td>RDB$RELATION_CONSTRAINTS</td>
<td>Definitions of all table-level constraints</td>
</tr>
<tr>
<td>RDB$RELATION_FIELDS</td>
<td>Top-level definitions of table columns</td>
</tr>
</tbody>
</table>
RDB$ROLES
Role definitions

RDB$SECURITY_CLASSES
Access control lists

RDB$TRANSACTIONS
State of multi-database transactions

RDB$TRIGGERS
Trigger definitions

RDB$TRIGGER_MESSAGES
Trigger messages

RDB$TYPES
Definitions of enumerated data types

RDB$USER_PRIVILEGES
SQL privileges granted to system users

RDB$VIEW_RELATIONS
Tables that are referred to in view definitions: one record for each table in a view

RDB$AUTH_MAPPING
RDB$AUTH_MAPPING stores authentication and other security mappings.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$MAP_NAME</td>
<td>CHAR(31)</td>
<td>Name of the mapping</td>
</tr>
<tr>
<td>RDB$MAP_USING</td>
<td>CHAR(1)</td>
<td>Using definition:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>P - plugin (specific or any)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S - any plugin serverwide</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M - mapping</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* - any method</td>
</tr>
<tr>
<td>RDB$MAP_PLUGIN</td>
<td>CHAR(31)</td>
<td>Mapping applies for authentication information from this specific plugin</td>
</tr>
<tr>
<td>RDB$MAP_DB</td>
<td>CHAR(31)</td>
<td>Mapping applies for authentication information from this specific database</td>
</tr>
<tr>
<td>RDB$MAP_FROM_TYPE</td>
<td>CHAR(31)</td>
<td>The type of authentication object (defined by plugin) to map from, or * for any type</td>
</tr>
<tr>
<td>RDB$MAP_FROM</td>
<td>CHAR(255)</td>
<td>The name of the authentication object to map from</td>
</tr>
</tbody>
</table>
## Appendix D: System Tables

### RDB$MAP_TO_TYPE

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$MAP_TO_TYPE</td>
<td>SMALLINT</td>
<td>The type to map to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - USER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - ROLE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$MAP_TO</td>
<td>CHAR(31)</td>
<td>The name to map to</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - user-defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 or higher - system-defined</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Optional description of the mapping</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(comment)</td>
</tr>
</tbody>
</table>

### RDB$BACKUP_HISTORY

RDB$BACKUP_HISTORY stores the history of backups performed using the nBackup utility.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$BACKUP_ID</td>
<td>INTEGER</td>
<td>The identifier assigned by the engine</td>
</tr>
<tr>
<td>RDB$TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>Backup date and time</td>
</tr>
<tr>
<td>RDB$BACKUP_LEVEL</td>
<td>INTEGER</td>
<td>Backup level</td>
</tr>
<tr>
<td>RDB$GUID</td>
<td>CHAR(38)</td>
<td>Unique identifier</td>
</tr>
<tr>
<td>RDB$SCN</td>
<td>INTEGER</td>
<td>System (scan) number</td>
</tr>
<tr>
<td>RDB$FILE_NAME</td>
<td>VARCHAR(255)</td>
<td>Full path and file name of backup file</td>
</tr>
</tbody>
</table>

### RDB$CHARACTER_SETS

RDB$CHARACTER_SETS names and describes the character sets available in the database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$CHARACTER_SET_NAME</td>
<td>CHAR(31)</td>
<td>Character set name</td>
</tr>
<tr>
<td>RDB$FORM_OF_USE</td>
<td>CHAR(31)</td>
<td>Not used</td>
</tr>
<tr>
<td>RDB$NUMBER_OF_CHARACTERS</td>
<td>INTEGER</td>
<td>The number of characters in the set. Not used for existing character sets</td>
</tr>
<tr>
<td>RDB$DEFAULT_COLLATE_NAME</td>
<td>CHAR(31)</td>
<td>The name of the default collation sequence for the character set</td>
</tr>
<tr>
<td>RDB$CHARACTER_SET_ID</td>
<td>SMALLINT</td>
<td>Unique character set identifier</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>System flag; value is 1 if the character set is defined in the system when the database is created; value is 0 for a user-defined character set</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Could store text description of the character set</td>
</tr>
<tr>
<td>RDB$FUNCTION_NAME</td>
<td>CHAR(31)</td>
<td>For a user-defined character set that is accessed via an external function, the name of the external function</td>
</tr>
<tr>
<td>RDB$BYTES_PER_CHARACTER</td>
<td>SMALLINT</td>
<td>The maximum number of bytes representing one character</td>
</tr>
<tr>
<td>RDB$SECURITY_CLASS</td>
<td>CHAR(31)</td>
<td>May reference a security class defined in the table RDB$SECURITY_CLASSES, in order to apply access control limits to all users of this character set</td>
</tr>
<tr>
<td>RDB$OWNER_NAME</td>
<td>CHAR(31)</td>
<td>The user name of the user who created the character set originally</td>
</tr>
</tbody>
</table>

**RDB$CHECK_CONSTRAINTS**

RDB$CHECK_CONSTRAINTS provides the cross references between the names of system-generated triggers for constraints and the names of the associated constraints (NOT NULL constraints, CHECK constraints and the ON UPDATE and ON DELETE clauses in foreign key constraints).

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$CONSTRAINT_NAME</td>
<td>CHAR(31)</td>
<td>Constraint name, defined by the user or automatically generated by the system</td>
</tr>
<tr>
<td>RDB$TRIGGER_NAME</td>
<td>CHAR(31)</td>
<td>For a CHECK constraint, it is the name of the trigger that enforces this constraint. For a NOT NULL constraint, it is the name of the table the constraint is applied to. For a foreign key constraint, it is the name of the trigger that enforces the ON UPDATE, ON DELETE clauses.</td>
</tr>
</tbody>
</table>

**RDB$COLLATIONS**

RDB$COLLATIONS stores collation sequences for all character sets.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$COLLATION_NAME</td>
<td>CHAR(31)</td>
<td>Collation sequence name</td>
</tr>
<tr>
<td>RDB$COLLATION_ID</td>
<td>SMALLINT</td>
<td>Collation sequence identifier. Together with the character set identifier, it is a unique collation sequence identifier</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>RDB$CHARACTER_SET_ID</code></td>
<td>SMALLINT</td>
<td>Character set identifier. Together with the collection sequence identifier, it is a unique identifier</td>
</tr>
<tr>
<td><code>RDB$COLLATION_ATTRIBUTES</code></td>
<td>SMALLINT</td>
<td>Collation attributes. It is a bit mask where the first bit shows whether trailing spaces should be taken into account in collations (0 - NO PAD; 1 - PAD SPACE); the second bit shows whether the collation is case-sensitive (0 - CASE SENSITIVE, 1 - CASE INSENSITIVE); the third bit shows whether the collation is accent-sensitive (0 - ACCENT SENSITIVE, 1 - ACCENT SENSITIVE). Thus, the value of 5 means that the collation does not take into account trailing spaces and is accent-insensitive</td>
</tr>
<tr>
<td><code>RDB$SYSTEM_FLAG</code></td>
<td>SMALLINT</td>
<td>Flag; the value of 0 means it is user-defined; the value of 1 means it is system-defined</td>
</tr>
<tr>
<td><code>RDB$DESCRIPTION</code></td>
<td>BLOB TEXT</td>
<td>Could store text description of the collation sequence</td>
</tr>
<tr>
<td><code>RDB$FUNCTION_NAME</code></td>
<td>CHAR(31)</td>
<td>Not currently used</td>
</tr>
<tr>
<td><code>RDB$BASE_COLLATION_NAME</code></td>
<td>CHAR(31)</td>
<td>The name of the base collation sequence for this collation sequence</td>
</tr>
<tr>
<td><code>RDB$SPECIFIC_ATTRIBUTES</code></td>
<td>BLOB TEXT</td>
<td>Describes specific attributes</td>
</tr>
<tr>
<td><code>RDB$SECURITY_CLASS</code></td>
<td>CHAR(31)</td>
<td>May reference a security class defined in the table <code>RDB$SECURITY_CLASSES</code>, in order to apply access control limits to all users of this collation</td>
</tr>
<tr>
<td><code>RDB$OWNER_NAME</code></td>
<td>CHAR(31)</td>
<td>The user name of the user who created the collation originally</td>
</tr>
</tbody>
</table>

**RDB$DATABASE**

`RDB$DATABASE` stores basic information about the database. It contains only one record.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RDB$DESCRIPTION</code></td>
<td>BLOB TEXT</td>
<td>Database comment text</td>
</tr>
<tr>
<td><code>RDB$RELATION_ID</code></td>
<td>SMALLINT</td>
<td>A number that steps up by one each time a table or view is added to the database</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$SECURITY_CLASS</td>
<td>CHAR(31)</td>
<td>The security class defined in RDB$SECURITY_CLASSES in order to apply access control limits common to the entire database</td>
</tr>
<tr>
<td>RDB$CHARACTER_SET_NAME</td>
<td>CHAR(31)</td>
<td>The name of the default character set for the database set in the DEFAULT CHARACTER_SET clause when the database is created. NULL for character set NONE.</td>
</tr>
<tr>
<td>RDB$LINGER</td>
<td>INTEGER</td>
<td>Number of seconds “delay” (established with the ALTER DATABASE SET LINGER statement) until the database file is closed after the last connection to this database is closed (in SuperServer). NULL if no delay is set.</td>
</tr>
</tbody>
</table>

**RDB$DB_CREATORS**

RDB$DB_CREATORS contains a list of users granted the CREATE DATABASE privilege when using the specified database as a security database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$USER</td>
<td>CHAR(31)</td>
<td>User or role name</td>
</tr>
<tr>
<td>RDB$USER_TYPE</td>
<td>SMALLINT</td>
<td>Type of user</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 - user</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 - role</td>
</tr>
</tbody>
</table>

**RDB$DEPENDENCIES**

RDB$DEPENDENCIES stores the dependencies between database objects.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$DEPENDENT_NAME</td>
<td>CHAR(31)</td>
<td>The name of the view, procedure, trigger, CHECK constraint or computed column the dependency is defined for, i.e., the dependent object</td>
</tr>
<tr>
<td>RDB$DEPENDED_ON_NAME</td>
<td>CHAR(31)</td>
<td>The name of the object that the defined object — the table, view, procedure, trigger, CHECK constraint or computed column — depends on</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$FIELD_NAME</td>
<td>CHAR(31)</td>
<td>The column name in the depended-on object that is referred to by the dependent view, procedure, trigger, CHECK constraint or computed column</td>
</tr>
<tr>
<td>RDB$DEPENDENT_TYPE</td>
<td>SMALLINT</td>
<td>Identifies the type of the dependent object:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - table</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - view</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - computed column</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 - CHECK constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 - procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 - index expression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 - exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 - user</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 - column</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 - index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15 - stored function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18 - package header</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19 - package body</td>
</tr>
<tr>
<td>RDB$DEPENDED_ON_TYPE</td>
<td>SMALLINT</td>
<td>Identifies the type of the object depended on:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - table (or a column in it)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - view</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - computed column</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 - CHECK constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 - procedure (or its parameter(s))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 - index expression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 - exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 - user</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 - column</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 - index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14 - generator (sequence)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15 - UDF or stored function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17 - collation 18 - package header</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19 - package body</td>
</tr>
<tr>
<td>RDB$PACKAGE_NAME</td>
<td>CHAR(31)</td>
<td>The package of a procedure or function for which this describes the dependency.</td>
</tr>
</tbody>
</table>
**RDB$EXCEPTIONS**

RDB$EXCEPTIONS stores custom database exceptions.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$EXCEPTION_NAME</td>
<td>CHAR(31)</td>
<td>Custom exception name</td>
</tr>
<tr>
<td>RDB$EXCEPTION_NUMBER</td>
<td>INTEGER</td>
<td>The unique number of the exception assigned by the system</td>
</tr>
<tr>
<td>RDB$MESSAGE</td>
<td>VARCHAR(1021)</td>
<td>Exception message text</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Could store text description of the exception</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - user-defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 or higher - system-defined</td>
</tr>
<tr>
<td>RDB$SECURITY_CLASS</td>
<td>CHAR(31)</td>
<td>May reference a security class defined in the table RDB$SECURITY_CLASSES, in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order to apply access control limits to all users of this exception</td>
</tr>
<tr>
<td>RDB$OWNER_NAME</td>
<td>CHAR(31)</td>
<td>The user name of the user who created the exception originally</td>
</tr>
</tbody>
</table>

**RDB$FIELDS**

RDB$FIELDS stores definitions of columns and domains, both system and custom. This is where the detailed data attributes are stored for all columns.

The column RDB$FIELDS.RDB$FIELD_NAME links to RDB$RELATION_FIELDS.RDB$FIELD_SOURCE, not to RDB$RELATION_FIELDS.RDB$FIELD_NAME.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$FIELD_NAME</td>
<td>CHAR(31)</td>
<td>The unique name of the domain created by the user or of the domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>automatically built for the table column by the system. System-created</td>
</tr>
<tr>
<td></td>
<td></td>
<td>domain names start with the “RDB$” prefix</td>
</tr>
<tr>
<td>RDB$QUERY_NAME</td>
<td>CHAR(31)</td>
<td>Not used</td>
</tr>
<tr>
<td>RDB$VALIDATION_BLR</td>
<td>BLOB BLR</td>
<td>The binary language representation (BLR) of the SQL expression specifying</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the check of the CHECK value in the domain</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$VALIDATION_SOURCE</td>
<td>BLOB TEXT</td>
<td>The original source text in the SQL language specifying the check of the CHECK value</td>
</tr>
<tr>
<td>RDB$COMPUTED_BLR</td>
<td>BLOB BLR</td>
<td>The binary language representation (BLR) of the SQL expression the database server uses for evaluation when accessing a COMPUTED BY column</td>
</tr>
<tr>
<td>RDB$COMPUTED_SOURCE</td>
<td>BLOB TEXT</td>
<td>The original source text of the expression that defines a COMPUTED BY column</td>
</tr>
<tr>
<td>RDB$DEFAULT_VALUE</td>
<td>BLOB BLR</td>
<td>The default value, if any, for the field or domain, in binary language representation (BLR)</td>
</tr>
<tr>
<td>RDB$DEFAULT_SOURCE</td>
<td>BLOB TEXT</td>
<td>The default value in the source code, as an SQL constant or expression</td>
</tr>
<tr>
<td>RDB$FIELD_LENGTH</td>
<td>SMALLINT</td>
<td>Column size in bytes. BOOLEAN occupies 1 byte. FLOAT, DATE, TIME, INTEGER occupy 4 bytes. DOUBLE PRECISION, BIGINT, TIMESTAMP and BLOB identifier occupy 8 bytes. For the CHAR and VARCHAR data types, the column stores the maximum number of bytes specified when a string domain (column) is defined</td>
</tr>
<tr>
<td>RDB$FIELD_SCALE</td>
<td>SMALLINT</td>
<td>The negative number that specifies the scale for DECIMAL and NUMERIC columns — the number of digits after the decimal point</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$FIELD_TYPE</td>
<td>SMALLINT</td>
<td>Data type code for the column:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 - SMALLINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 - INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 - FLOAT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12 - DATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 - TIME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14 - CHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 - BIGINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23 - BOOLEAN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>27 - DOUBLE PRECISION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>35 - TIMESTAMP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>37 - VARCHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>261 - BLOB</td>
</tr>
</tbody>
</table>

Codes for DECIMAL and NUMERIC are the same as for the integer types used to store them.
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$FIELD_SUB_TYPE</td>
<td>SMALLINT</td>
<td>Specifies the subtype for the BLOB data type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - untyped (binary)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - BLR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - access control list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 - reserved for future use</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 - encoded table metadata description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 - for storing the details of a cross-database transaction that ends abnormally</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 - external file description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 - debug information (for PSQL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt; 0 - user-defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies for the CHAR data type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - untyped data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - fixed binary data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies the particular data type for the integer data types (SMALLINT, INTEGER, BIGINT) and for fixed-point numbers (NUMERIC, DECIMAL):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 or NULL - the data type matches the value in the RDB$FIELD_TYPE field</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - NUMERIC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - DECIMAL</td>
</tr>
<tr>
<td>RDB$MISSING_VALUE</td>
<td>BLOB BLR</td>
<td>Not used</td>
</tr>
<tr>
<td>RDB$MISSING_SOURCE</td>
<td>BLOB TEXT</td>
<td>Not used</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Any domain (table column) comment text</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Flag: the value of 1 means the domain is automatically created by the system, the value of 0 means that the domain is defined by the user</td>
</tr>
<tr>
<td>RDB$QUERY_HEADER</td>
<td>BLOB TEXT</td>
<td>Not used</td>
</tr>
<tr>
<td>RDB$SEGMENT_LENGTH</td>
<td>SMALLINT</td>
<td>Specifies the length of the BLOB buffer in bytes for BLOB columns. Stores NULL for all other data types</td>
</tr>
<tr>
<td>RDB$EDIT_STRING</td>
<td>VARCHAR(127)</td>
<td>Not used</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$EXTERNAL_LENGTH</td>
<td>SMALLINT</td>
<td>The length of the column in bytes if it belongs to an external table. Always NULL for regular tables</td>
</tr>
<tr>
<td>RDB$EXTERNAL_SCALE</td>
<td>SMALLINT</td>
<td>The scale factor of an integer-type field in an external table; represents the power of 10 by which the integer is multiplied</td>
</tr>
<tr>
<td>RDB$EXTERNAL_TYPE</td>
<td>SMALLINT</td>
<td>The data type of the field as it is represented in an external table:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 - SMALLINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 - INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 - FLOAT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12 - DATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 - TIME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14 - CHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 - BIGINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23 - BOOLEAN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>27 - DOUBLE PRECISION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>35 - TIMESTAMP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>37 - VARCHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>261 - BLOB</td>
</tr>
<tr>
<td>RDB$DIMENSIONS</td>
<td>SMALLINT</td>
<td>Defines the number of dimensions in an array if the column is defined as an array. Always NULL for columns that are not arrays</td>
</tr>
<tr>
<td>RDB$NULL_FLAG</td>
<td>SMALLINT</td>
<td>Specifies whether the column can take an empty value (the field will contain NULL) or not (the field will contain the value of 1)</td>
</tr>
<tr>
<td>RDB$CHARACTER_LENGTH</td>
<td>SMALLINT</td>
<td>The length of CHAR or VARCHAR columns in characters (not in bytes)</td>
</tr>
<tr>
<td>RDB$COLLATION_ID</td>
<td>SMALLINT</td>
<td>The identifier of the collation sequence for a character column or domain. If it is not defined, the value of the field will be 0</td>
</tr>
<tr>
<td>RDB$CHARACTER_SET_ID</td>
<td>SMALLINT</td>
<td>The identifier of the character set for a character column, BLOB TEXT column or domain</td>
</tr>
</tbody>
</table>
### Column Name | Data Type | Description
--- | --- | ---
**RDB$FIELD_PRECISION** | SMALLINT | Specifies the total number of digits for the fixed-point numeric data type (DECIMAL and NUMERIC). The value is 0 for the integer data types, NULL is for other data types.

**RDB$SECURITY_CLASS** | CHAR(31) | May reference a security class defined in the table RDB$SECURITY_CLASSES, in order to apply access control limits to all users of this domain.

**RDB$OWNER_NAME** | CHAR(31) | The user name of the user who created the domain originally.

---

**RDB$FIELD_DIMENSIONS**

RDB$FIELD_DIMENSIONS stores the dimensions of array columns.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RDB$FIELD_NAME</strong></td>
<td>CHAR(31)</td>
<td>The name of the array column. It must be present in the RDB$FIELD_NAME field of the RDB$FIELDS table.</td>
</tr>
<tr>
<td><strong>RDB$DIMENSION</strong></td>
<td>SMALLINT</td>
<td>Identifies one dimension in the array column. The numbering of dimensions starts with 0.</td>
</tr>
<tr>
<td><strong>RDB$LOWER_BOUND</strong></td>
<td>INTEGER</td>
<td>The lower bound of this dimension.</td>
</tr>
<tr>
<td><strong>RDB$UPPER_BOUND</strong></td>
<td>INTEGER</td>
<td>The upper bound of this dimension.</td>
</tr>
</tbody>
</table>

---

**RDB$FILES**

RDB$FILES stores information about secondary files and shadow files.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RDB$FILE_NAME</strong></td>
<td>VARCHAR(255)</td>
<td>The full path to the file and the name of either</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• the database secondary file in a multi-file database, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• the shadow file</td>
</tr>
<tr>
<td><strong>RDB$FILE_SEQUENCE</strong></td>
<td>SMALLINT</td>
<td>The sequential number of the secondary file in a sequence or of the shadow file in a shadow file set.</td>
</tr>
<tr>
<td><strong>RDB$FILE_START</strong></td>
<td>INTEGER</td>
<td>The initial page number in the secondary file or shadow file.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$FILE_LENGTH</td>
<td>INTEGER</td>
<td>File length in database pages</td>
</tr>
<tr>
<td>RDB$FILE_FLAGS</td>
<td>SMALLINT</td>
<td>For internal use</td>
</tr>
<tr>
<td>RDB$SHADOW_NUMBER</td>
<td>SMALLINT</td>
<td>Shadow set number. If the row describes a database secondary file, the field will be NULL or its value will be 0</td>
</tr>
</tbody>
</table>

**RDB$FILTERS**

RDB$FILTERS stores information about BLOB filters.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$FUNCTION_NAME</td>
<td>CHAR(31)</td>
<td>The unique identifier of the BLOB filter</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Documentation about the BLOB filter and the two subtypes it is used with, written by the user</td>
</tr>
<tr>
<td>RDB$MODULE_NAME</td>
<td>VARCHAR(255)</td>
<td>The name of the dynamic library or shared object where the code of the BLOB filter is located</td>
</tr>
<tr>
<td>RDB$ENTRYPOINT</td>
<td>CHAR(255)</td>
<td>The exported name of the BLOB filter in the filter library. Note, this is often not the same as RDB$FUNCTION_NAME, which is the identifier with which the BLOB filter is declared to the database</td>
</tr>
<tr>
<td>RDB$INPUT_SUB_TYPE</td>
<td>SMALLINT</td>
<td>The BLOB subtype of the data to be converted by the function</td>
</tr>
<tr>
<td>RDB$OUTPUT_SUB_TYPE</td>
<td>SMALLINT</td>
<td>The BLOB subtype of the converted data</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Flag indicating whether the filter is user-defined or internally defined:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - user-defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 or greater - internally defined</td>
</tr>
<tr>
<td>RDB$SECURITY_CLASS</td>
<td>CHAR(31)</td>
<td>May reference a security class defined in the table RDB$SECURITY_CLASSES, in order to apply access control limits to all users of this filter</td>
</tr>
<tr>
<td>RDB$OWNER_NAME</td>
<td>CHAR(31)</td>
<td>The user name of the user who created the filter originally</td>
</tr>
</tbody>
</table>

**RDB$FORMATS**

RDB$FORMATS stores information about changes in tables. Each time any metadata change to a table is committed, it gets a new format number. When the format number of any table reaches 255, the
entire database becomes inoperable. To return to normal, the database must be backed up with the\n`gbak` utility and restored from that backup copy.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$RELATION_ID</td>
<td>SMALLINT</td>
<td>Table or view identifier</td>
</tr>
<tr>
<td>RDB$FORMAT</td>
<td>SMALLINT</td>
<td>Table format identifier — maximum 255. The critical time comes when this</td>
</tr>
<tr>
<td></td>
<td></td>
<td>number approaches 255 for any table or view</td>
</tr>
<tr>
<td>RDB$DESCRIPTOR</td>
<td>BLOB FORMAT</td>
<td>Stores column names and data attributes as BLOB, as they were at the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time the format record was created</td>
</tr>
</tbody>
</table>

### RDB$FUNCTIONS

RDB$FUNCTIONS stores the information needed by the engine about stored functions and external functions (user-defined functions, UDFs).

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$FUNCTION_NAME</td>
<td>CHAR(31)</td>
<td>The unique (declared) name of the external function</td>
</tr>
<tr>
<td>RDB$FUNCTION_TYPE</td>
<td>SMALLINT</td>
<td>Not currently used</td>
</tr>
<tr>
<td>RDB$QUERY_NAME</td>
<td>CHAR(31)</td>
<td>Not currently used</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Any text with comments related to the external function</td>
</tr>
<tr>
<td>RDB$MODULE_NAME</td>
<td>VARCHAR(255)</td>
<td>The name of the dynamic library or shared object where the code of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>external function is located</td>
</tr>
<tr>
<td>RDB$ENTRYPOINT</td>
<td>CHAR(255)</td>
<td>The exported name of the external function in the function library. Note,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>this is often not the same as RDB$FUNCTION_NAME, which is the identifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>with which the external function is declared to the database</td>
</tr>
<tr>
<td>RDB$RETURN_ARGUMENT</td>
<td>SMALLINT</td>
<td>The position number of the returned argument in the list of parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>corresponding to input arguments</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Flag indicating whether the filter is user-defined or internally defined:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - user-defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - internally defined</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$ENGINE_NAME</td>
<td>CHAR(31)</td>
<td>Engine for external functions. 'UDR' for UDR functions. NULL for legacy UDF or PSQL functions</td>
</tr>
<tr>
<td>RDB$PACKAGE_NAME</td>
<td>CHAR(31)</td>
<td>Package that contains this function (or NULL)</td>
</tr>
<tr>
<td>RDB$PRIVATE_FLAG</td>
<td>SMALLINT</td>
<td>NULL for normal (top-level) functions, 0 for package function defined in the header, 1 for package function only defined in the package body.</td>
</tr>
<tr>
<td>RDB$FUNCTION_SOURCE</td>
<td>BLOB TEXT</td>
<td>The PSQL sourcecode of the function</td>
</tr>
<tr>
<td>RDB$FUNCTION_ID</td>
<td>SMALLINT</td>
<td>Unique identifier of the function</td>
</tr>
<tr>
<td>RDB$FUNCTION_BLR</td>
<td>BLOB BLR</td>
<td>The binary language representation (BLR) of the function code (PSQL function only)</td>
</tr>
<tr>
<td>RDB$VALID_BLR</td>
<td>SMALLINT</td>
<td>Indicates whether the source PSQL of the stored procedure remains valid after the latest ALTER FUNCTION modification</td>
</tr>
<tr>
<td>RDB$DEBUG_INFO</td>
<td>BLOB DEBUG_INFORMATION</td>
<td>Contains debugging information about variables used in the function (PSQL function only)</td>
</tr>
<tr>
<td>RDB$SECURITY_CLASS</td>
<td>CHAR(31)</td>
<td>May reference a security class defined in the table RDB$SECURITY_CLASSES, in order to apply access control limits to all users of this function</td>
</tr>
<tr>
<td>RDB$OWNER_NAME</td>
<td>CHAR(31)</td>
<td>The user name of the user who created the function originally</td>
</tr>
<tr>
<td>RDB$LEGACY_FLAG</td>
<td>SMALLINT</td>
<td>The legacy style attribute of the function. 1 if the function is described in legacy style (DECLARE EXTERNAL FUNCTION), otherwise CREATE FUNCTION.</td>
</tr>
<tr>
<td>RDB$DETERMINISTIC_FLAG</td>
<td>SMALLINT</td>
<td>Deterministic flag. 1 if function is deterministic</td>
</tr>
</tbody>
</table>

**RDB$FUNCTION_ARGUMENTS**

RDB$FUNCTION_ARGUMENTS stores the parameters of external functions and their attributes.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$FUNCTION_NAME</td>
<td>CHAR(31)</td>
<td>The unique name (declared identifier) of the external function</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$ARGUMENT_POSITION</td>
<td>SMALLINT</td>
<td>The position of the argument in the list of arguments</td>
</tr>
<tr>
<td>RDB$MECHANISM</td>
<td>SMALLINT</td>
<td>Flag: how this argument is passed:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - by value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - by reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - by descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - by BLOB descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 - by scalar array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 - by reference with null</td>
</tr>
<tr>
<td>RDB$FIELD_TYPE</td>
<td>SMALLINT</td>
<td>Data type code defined for the column:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 - SMALLINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 - INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12 - DATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 - TIME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14 - CHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 - BIGINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23 - BOOLEAN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>27 - DOUBLE PRECISION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>35 - TIMESTAMP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>37 - VARCHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40 - CSTRING (null-terminated text)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>45 - BLOB_ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>261 - BLOB</td>
</tr>
<tr>
<td>RDB$FIELD_SCALE</td>
<td>SMALLINT</td>
<td>The scale of an integer or a fixed-point argument. It is an exponent of 10</td>
</tr>
<tr>
<td>RDB$FIELD_LENGTH</td>
<td>SMALLINT</td>
<td>Argument length in bytes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BOOLEAN = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SMALLINT = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER = 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATE = 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIME = 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BIGINT = 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DOUBLE PRECISION = 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIMESTAMP = 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BLOB_ID = 8</td>
</tr>
<tr>
<td>RDB$FIELD_SUB_TYPE</td>
<td>SMALLINT</td>
<td>Stores the BLOB subtype for an argument of a BLOB data type</td>
</tr>
<tr>
<td>RDB$CHARACTER_SET_ID</td>
<td>SMALLINT</td>
<td>The identifier of the character set for a character argument</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$FIELD_PRECISION</td>
<td>SMALLINT</td>
<td>The number of digits of precision available for the data type of the argument</td>
</tr>
<tr>
<td>RDB$CHARACTER_LENGTH</td>
<td>SMALLINT</td>
<td>The length of a CHAR or VARCHAR argument in characters (not in bytes)</td>
</tr>
<tr>
<td>RDB$PACKAGE_NAME</td>
<td>CHAR(31)</td>
<td>Package name of the function (or NULL for a top-level function)</td>
</tr>
<tr>
<td>RDB$ARGUMENT_NAME</td>
<td>CHAR(31)</td>
<td>Parameter name</td>
</tr>
<tr>
<td>RDB$FIELD_SOURCE</td>
<td>CHAR(31)</td>
<td>The name of the user-created domain, when a domain is referenced instead of a data type. If the name starts with the prefix “RDB$”, it is the name of the domain automatically generated by the system for the parameter.</td>
</tr>
<tr>
<td>RDB$DEFAULT_VALUE</td>
<td>BLOB BLR</td>
<td>The default value for the parameter, in the binary language representation (BLR)</td>
</tr>
<tr>
<td>RDB$DEFAULT_SOURCE</td>
<td>BLOB TEXT</td>
<td>The default value for the parameter, in PSQL code</td>
</tr>
<tr>
<td>RDB$COLLATION_ID</td>
<td>SMALLINT</td>
<td>The identifier of the collation sequence used for a character parameter</td>
</tr>
<tr>
<td>RDB$NULL_FLAG</td>
<td>SMALLINT</td>
<td>The flag indicating whether NULL is allowable</td>
</tr>
<tr>
<td>RDB$ARGUMENT_MECHANISM</td>
<td>SMALLINT</td>
<td>Parameter passing mechanism for non-legacy functions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - by value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - by reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - through a descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - via the BLOB descriptor</td>
</tr>
<tr>
<td>RDB$FIELD_NAME</td>
<td>CHAR(31)</td>
<td>The name of the column the parameter references, if it was declared using TYPE OF COLUMN instead of a regular data type. Used in conjunction with RDB$RELATION_NAME (see next).</td>
</tr>
<tr>
<td>RDB$RELATION_NAME</td>
<td>CHAR(31)</td>
<td>The name of the table the parameter references, if it was declared using TYPE OF COLUMN instead of a regular data type</td>
</tr>
</tbody>
</table>
Appendix D: System Tables

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - user-defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 or higher - system-defined</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Optional description of the function argument (comment)</td>
</tr>
</tbody>
</table>

RDB$GENERATORS

RDB$GENERATORS stores generators (sequences) and keeps them up-to-date.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$GENERATOR_NAME</td>
<td>CHAR(31)</td>
<td>The unique name of the generator</td>
</tr>
<tr>
<td>RDB$GENERATOR_ID</td>
<td>SMALLINT</td>
<td>The unique identifier assigned to the generator by the system</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - user-defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 or greater - system-defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 - internal generator for identity column</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Could store comments related to the generator</td>
</tr>
<tr>
<td>RDB$SECURITY_CLASS</td>
<td>CHAR(31)</td>
<td>May reference a security class defined in the table RDB$SECURITY_CLASSES, in order to apply access control limits to all users of this generator</td>
</tr>
<tr>
<td>RDB$OWNER_NAME</td>
<td>CHAR(31)</td>
<td>The user name of the user who created the generator originally</td>
</tr>
<tr>
<td>RDB$INITIAL_VALUE</td>
<td>BIGINT</td>
<td>Stores the initial value (START WITH value) of the generator</td>
</tr>
<tr>
<td>RDB$GENERATOR_INCREMENT</td>
<td>INTEGER</td>
<td>Stores the increment of the value (INCREMENT BY value) of the generator</td>
</tr>
</tbody>
</table>

RDB$INDICES

RDB$INDICES stores definitions of both system- and user-defined indexes. The attributes of each column belonging to an index are stored in one row of the table RDB$INDEX_SEGMENTS.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$INDEX_NAME</td>
<td>CHAR(31)</td>
<td>The unique name of the index specified by the user or automatically generated by the system</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$RELATION_NAME</td>
<td>CHAR(31)</td>
<td>The name of the table to which the index belongs. It corresponds to an identifier in RDB$RELATION_NAME.RDB$RELATIONS</td>
</tr>
<tr>
<td>RDB$INDEX_ID</td>
<td>SMALLINT</td>
<td>The internal (system) identifier of the index</td>
</tr>
<tr>
<td>RDB$UNIQUE_FLAG</td>
<td>SMALLINT</td>
<td>Specifies whether the index is unique: 0 - not unique 1 - unique</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Could store comments concerning the index</td>
</tr>
<tr>
<td>RDB$SEGMENT_COUNT</td>
<td>SMALLINT</td>
<td>The number of segments (columns) in the index</td>
</tr>
<tr>
<td>RDB$INDEX_INACTIVE</td>
<td>SMALLINT</td>
<td>Indicates whether the index is currently active: 0 - active 1 - inactive</td>
</tr>
<tr>
<td>RDB$INDEX_TYPE</td>
<td>SMALLINT</td>
<td>Distinguishes between an ascending (0 or NULL) and descending index (1). Not used in databases created before Firebird 2.0; hence, indexes in upgraded databases are more likely to store NULL in this column</td>
</tr>
<tr>
<td>RDB$FOREIGN_KEY</td>
<td>CHAR(31)</td>
<td>The name of the primary or unique key index referenced by the foreign key backed by this index; NULL if this index is not used by a foreign key.</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Indicates whether the index is system-defined or user-defined: 0 - user-defined 1 or greater - system-defined</td>
</tr>
<tr>
<td>RDB$EXPRESSION_BLR</td>
<td>BLOB BLR</td>
<td>Expression for an expression index, written in the binary language representation (BLR), used for calculating the values for the index at runtime.</td>
</tr>
<tr>
<td>RDB$EXPRESSION_SOURCE</td>
<td>BLOB TEXT</td>
<td>The source code of the expression for an expression index</td>
</tr>
</tbody>
</table>
**RDB$STATISTICS**

Stores the last known selectivity of the entire index, calculated by execution of a `SET STATISTICS` statement over the index. It is also recalculated whenever the database is first opened by the server. The selectivity of each separate segment of the index is stored in `RDB$INDEX_SEGMENTS`.

**RDB$INDEX_SEGMENTS**

`RDB$INDEX_SEGMENTS` stores the segments (table columns) of indexes and their positions in the key. A separate row is stored for each column in an index.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$INDEX_NAME</td>
<td>CHAR(31)</td>
<td>The name of the index this segment is related to. The master record is RDB$INDICES.RDB$INDEX_NAME.</td>
</tr>
<tr>
<td>RDB$FIELD_NAME</td>
<td>CHAR(31)</td>
<td>The name of a column belonging to the index, corresponding to an identifier for the table and that column in RDB$RELATION_FIELDS.RDB$FIELD_NAME.</td>
</tr>
<tr>
<td>RDB$FIELD_POSITION</td>
<td>SMALLINT</td>
<td>The column position in the index. Positions are numbered left-to-right, starting at zero</td>
</tr>
<tr>
<td>RDB$STATISTICS</td>
<td>DOUBLE PRECISION</td>
<td>The last known (calculated) selectivity of this column in the index. The higher the number, the lower the selectivity.</td>
</tr>
</tbody>
</table>

**RDB$LOG_FILES**

`RDB$LOG_FILES` is not currently used.

**RDB$PACKAGES**

`RDB$PACKAGES` stores the definition (header and body) of SQL packages.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$PACKAGE_NAME</td>
<td>CHAR(31)</td>
<td>Name of the package</td>
</tr>
<tr>
<td>RDB$PACKAGE_HEADER_SOURCE</td>
<td>BLOB TEXT</td>
<td>The PSQL sourcecode of the package header</td>
</tr>
<tr>
<td>RDB$PACKAGE_BODY_SOURCE</td>
<td>BLOB TEXT</td>
<td>The PSQL sourcecode of the package body</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$VALID_BODY_FLAG</td>
<td>SMALLINT</td>
<td>Indicates whether the body of the package is still valid. NULL or 0 indicates the body is not valid.</td>
</tr>
<tr>
<td>RDB$SECURITY_CLASS</td>
<td>CHAR(31)</td>
<td>May reference a security class defined in the table RDB$SECURITY_CLASSES, in order to apply access control limits to all users of this package.</td>
</tr>
<tr>
<td>RDB$OWNER_NAME</td>
<td>CHAR(31)</td>
<td>The user name of the user who created the package originally.</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Flag: 0 - user-defined 1 or higher - system-defined</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Optional description of the package (comment)</td>
</tr>
</tbody>
</table>

### RDB$PAGES

RDB$PAGES stores and maintains information about database pages and their usage.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$PAGE_NUMBER</td>
<td>INTEGER</td>
<td>The unique number of a physically created database page</td>
</tr>
<tr>
<td>RDB$RELATION_ID</td>
<td>SMALLINT</td>
<td>The identifier of the table to which the page is allocated</td>
</tr>
<tr>
<td>RDB$PAGE_SEQUENCE</td>
<td>INTEGER</td>
<td>The number of the page in the sequence of all pages allocated to this table</td>
</tr>
<tr>
<td>RDB$PAGE_TYPE</td>
<td>SMALLINT</td>
<td>Indicates the page type (data, index, BLOB, etc.). For system use</td>
</tr>
</tbody>
</table>

### RDB$PROCEDURES

RDB$PROCEDURES stores the definitions of stored procedures, including their PSQL source code and the binary language representation (BLR) of it. The next table, RDB$PROCEDURE_PARAMETERS, stores the definitions of input and output parameters.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$PROCEDURE_NAME</td>
<td>CHAR(31)</td>
<td>Stored procedure name (identifier)</td>
</tr>
<tr>
<td>RDB$PROCEDURE_ID</td>
<td>SMALLINT</td>
<td>The procedure’s unique, system-generated identifier</td>
</tr>
<tr>
<td>RDB$PROCEDURE_INPUTS</td>
<td>SMALLINT</td>
<td>Indicates the number of input parameters. NULL if there are none</td>
</tr>
</tbody>
</table>

Appendix D: System Tables
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$PROCEDURE_OUTPUTS</td>
<td>SMALLINT</td>
<td>Indicates the number of output parameters. NULL if there are none</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Any text comments related to the procedure</td>
</tr>
<tr>
<td>RDB$PROCEDURE_SOURCE</td>
<td>BLOB TEXT</td>
<td>The PSQL source code of the procedure</td>
</tr>
<tr>
<td>RDB$PROCEDURE_BLR</td>
<td>BLOB BLR</td>
<td>The binary language representation (BLR) of the procedure code</td>
</tr>
<tr>
<td>RDB$SECURITY_CLASS</td>
<td>CHAR(31)</td>
<td>May point to the security class defined in the system table RDB$SECURITY_CLASSES in order to apply access control limits</td>
</tr>
<tr>
<td>RDB$OWNER_NAME</td>
<td>CHAR(31)</td>
<td>The user name of the procedure’s Owner — the user who was CURRENT_USER when the procedure was first created. It may or may not be the user name of the author.</td>
</tr>
<tr>
<td>RDB$RUNTIME</td>
<td>BLOB</td>
<td>A metadata description of the procedure, used internally for optimization</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Indicates whether the procedure is defined by a user (value 0) or by the system (a value of 1 or greater)</td>
</tr>
<tr>
<td>RDB$PROCEDURE_TYPE</td>
<td>SMALLINT</td>
<td>Procedure type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - selectable stored procedure (contains a SUSPEND statement)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - executable stored procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NULL - not known *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* for procedures created before Firebird 1.5</td>
</tr>
<tr>
<td>RDB$VALID_BLR</td>
<td>SMALLINT</td>
<td>Indicates whether the source PSQL of the stored procedure remains valid after the latest ALTER PROCEDURE modification</td>
</tr>
<tr>
<td>RDB$DEBUG_INFO</td>
<td>BLOB DEBUG_INFORMATION</td>
<td>Contains debugging information about variables used in the stored procedure</td>
</tr>
<tr>
<td>RDB$ENGINE_NAME</td>
<td>CHAR(31)</td>
<td>Engine for external functions. 'UDR' for UDR procedures. NULL for PSQL stored procedures</td>
</tr>
</tbody>
</table>
### RDB$ENTRYPOINT

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$ENTRYPOINT</td>
<td>CHAR(255)</td>
<td>The exported name of the external function in the procedure library. Note, this is often not the same as RDB$PROCEDURE_NAME, which is the identifier with which the external stored procedure is declared to the database</td>
</tr>
</tbody>
</table>

### RDB$PACKAGE_NAME

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$PACKAGE_NAME</td>
<td>CHAR(31)</td>
<td>Package name of the procedure (or NULL for a top-level stored procedure)</td>
</tr>
</tbody>
</table>

### RDB$PRIVATE_FLAG

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$PRIVATE_FLAG</td>
<td>SMALLINT</td>
<td>NULL for normal (top-level) stored procedures, 0 for package procedures defined in the header, 1 for package procedures only defined in the package body</td>
</tr>
</tbody>
</table>

### RDB$PROCEDURE_PARAMETERS

RDB$PROCEDURE_PARAMETERS stores the parameters of stored procedures and their attributes. It holds one row for each parameter.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$PARAMETER_NAME</td>
<td>CHAR(31)</td>
<td>Parameter name</td>
</tr>
<tr>
<td>RDB$PROCEDURE_NAME</td>
<td>CHAR(31)</td>
<td>The name of the procedure where the parameter is defined</td>
</tr>
<tr>
<td>RDB$PARAMETER_NUMBER</td>
<td>SMALLINT</td>
<td>The sequential number of the parameter</td>
</tr>
<tr>
<td>RDB$PARAMETER_TYPE</td>
<td>SMALLINT</td>
<td>Indicates whether the parameter is for input (value 0) or output (value 1)</td>
</tr>
<tr>
<td>RDB$FIELD_SOURCE</td>
<td>CHAR(31)</td>
<td>The name of the user-created domain, when a domain is referenced instead of a data type. If the name starts with the prefix “RDB$”, it is the name of the domain automatically generated by the system for the parameter.</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Could store comments related to the parameter</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Indicates whether the parameter was defined by the system (value or greater) or by a user (value 0)</td>
</tr>
<tr>
<td>RDB$DEFAULT_VALUE</td>
<td>BLOB BLR</td>
<td>The default value for the parameter, in the binary language representation (BLR)</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$DEFAULT_SOURCE</td>
<td>BLOB TEXT</td>
<td>The default value for the parameter, in PSQL code</td>
</tr>
<tr>
<td>RDB$COLLATION_ID</td>
<td>SMALLINT</td>
<td>The identifier of the collation sequence used for a character parameter</td>
</tr>
<tr>
<td>RDB$NULL_FLAG</td>
<td>SMALLINT</td>
<td>The flag indicating whether NULL is allowable</td>
</tr>
<tr>
<td>RDB$PARAMETER_MECHANISM</td>
<td>SMALLINT</td>
<td>Flag: indicates how this parameter is passed:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - by value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - by reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - by descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - by BLOB descriptor</td>
</tr>
<tr>
<td>RDB$FIELD_NAME</td>
<td>CHAR(31)</td>
<td>The name of the column the parameter references, if it was declared using TYPE OF COLUMN instead of a regular data type. Used in conjunction with RDB$RELATION_NAME (see next).</td>
</tr>
<tr>
<td>RDB$RELATION_NAME</td>
<td>CHAR(31)</td>
<td>The name of the table the parameter references, if it was declared using TYPE OF COLUMN instead of a regular data type</td>
</tr>
<tr>
<td>RDB$PACKAGE_NAME</td>
<td>CHAR(31)</td>
<td>Package name of the procedure (or NULL for a top-level stored procedure)</td>
</tr>
</tbody>
</table>

**RDB$REF_CONSTRAINTS**

RDB$REF_CONSTRAINTS stores the attributes of the referential constraints — Foreign Key relationships and referential actions.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$CONSTRAINT_NAME</td>
<td>CHAR(31)</td>
<td>Foreign key constraint name, defined by the user or automatically generated by the system</td>
</tr>
<tr>
<td>RDB$CONST_NAME_UQ</td>
<td>CHAR(31)</td>
<td>The name of the primary or unique key constraint linked by the REFERENCES clause in the constraint definition</td>
</tr>
<tr>
<td>RDB$MATCH_OPTION</td>
<td>CHAR(7)</td>
<td>Not used. The current value is FULL in all cases</td>
</tr>
</tbody>
</table>
### Column Name | Data Type | Description
--- | --- | ---
RDB$UPDATE_RULE | CHAR(11) | Referential integrity actions applied to the foreign key record(s) when the primary (unique) key of the parent table is updated: RESTRICT, NO ACTION, CASCADE, SET NULL, SET DEFAULT
RDB$DELETE_RULE | CHAR(11) | Referential integrity actions applied to the foreign key record(s) when the primary (unique) key of the parent table is deleted: RESTRICT, NO ACTION, CASCADE, SET NULL, SET DEFAULT

### RDB$RELATIONS

RDB$RELATIONS stores the top-level definitions and attributes of all tables and views in the system.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$VIEW_BLR</td>
<td>BLOB BLR</td>
<td>Stores the query specification for a view, in the binary language representation (BLR). The field stores NULL for a table</td>
</tr>
<tr>
<td>RDB$VIEW_SOURCE</td>
<td>BLOB TEXT</td>
<td>Contains the original source text of the query for a view, in SQL language. User comments are included. The field stores NULL for a table</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Could store comments related to the table or view</td>
</tr>
<tr>
<td>RDB$RELATION_ID</td>
<td>SMALLINT</td>
<td>Internal identifier of the table or view</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>indicates whether the table or view is user-defined (value 0) or system-defined (value 1 or greater)</td>
</tr>
<tr>
<td>RDB$DBKEY_LENGTH</td>
<td>SMALLINT</td>
<td>The total length of the database key. For a table: 8 bytes. For a view, the length is 8 multiplied by the number of tables referenced by the view</td>
</tr>
<tr>
<td>RDB$FORMAT</td>
<td>SMALLINT</td>
<td>Internal use, points to the relation’s record in RDB$FORMATS — do not modify</td>
</tr>
<tr>
<td>RDB$FIELD_ID</td>
<td>SMALLINT</td>
<td>The field ID for the next column to be added. The number is not decremented when a column is dropped.</td>
</tr>
<tr>
<td>RDB$RELATION_NAME</td>
<td>CHAR(31)</td>
<td>Table or view name</td>
</tr>
</tbody>
</table>
### RDB$SECURITY_CLASS

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$SECURITY_CLASS</td>
<td>CHAR(31)</td>
<td>May reference a security class defined in the table RDB$SECURITY_CLASSES, in order to apply access control limits to all users of this table or view</td>
</tr>
</tbody>
</table>

### RDB$EXTERNAL_FILE

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$EXTERNAL_FILE</td>
<td>VARCHAR(255)</td>
<td>The full path to the external data file if the table is defined with the EXTERNAL FILE clause</td>
</tr>
</tbody>
</table>

### RDB$RUNTIME

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$RUNTIME</td>
<td>BLOB</td>
<td>Table metadata description, used internally for optimization</td>
</tr>
</tbody>
</table>

### RDB$EXTERNAL_DESCRIPTION

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$EXTERNAL_DESCRIPTION</td>
<td>BLOB</td>
<td>Could store comments related to the external file of an external table</td>
</tr>
</tbody>
</table>

### RDB$OWNER_NAME

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$OWNER_NAME</td>
<td>CHAR(31)</td>
<td>The user name of the user who created the table or view originally</td>
</tr>
</tbody>
</table>

### RDB$DEFAULT_CLASS

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$DEFAULT_CLASS</td>
<td>CHAR(31)</td>
<td>Default security class, used when a new column is added to the table</td>
</tr>
</tbody>
</table>

### RDB$FLAGS

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$FLAGS</td>
<td>SMALLINT</td>
<td>Internal flags</td>
</tr>
</tbody>
</table>

### RDB$RELATION_TYPE

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$RELATION_TYPE</td>
<td>SMALLINT</td>
<td>The type of the relation object being described: 0 - system or user-defined table 1 - view 2 - external table 3 - monitoring table 4 - connection-level GTT (PRESERVE ROWS) 5 - transaction-level GTT (DELETE ROWS)</td>
</tr>
</tbody>
</table>

### RDB$RELATION_CONSTRAINTS

RDB$RELATION_CONSTRAINTS stores the definitions of all table-level constraints: primary, unique, foreign key, CHECK, NOT NULL constraints.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$CONSTRAINT_NAME</td>
<td>CHAR(31)</td>
<td>The name of the table-level constraint defined by the user, or otherwise automatically generated by the system</td>
</tr>
</tbody>
</table>

### RDB$CONSTRAINT_TYPE

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$CONSTRAINT_TYPE</td>
<td>CHAR(11)</td>
<td>The name of the constraint type: PRIMARY KEY, UNIQUE, FOREIGN KEY, CHECK or NOT NULL</td>
</tr>
</tbody>
</table>

### RDB$RELATION_NAME

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$RELATION_NAME</td>
<td>CHAR(31)</td>
<td>The name of the table this constraint applies to</td>
</tr>
</tbody>
</table>

### RDB$DEFERRABLE

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$DEFERRABLE</td>
<td>CHAR(3)</td>
<td>Currently NO in all cases: Firebird does not yet support deferrable constraints</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$INITIALLY_DEFERRED</td>
<td>CHAR(3)</td>
<td>Currently NO in all cases</td>
</tr>
<tr>
<td>RDB$INDEX_NAME</td>
<td>CHAR(31)</td>
<td>The name of the index that supports this constraint. For a CHECK or a NOT NULL constraint, it is NULL.</td>
</tr>
</tbody>
</table>

**RDB$RELATION_FIELDS**

RDB$RELATION_FIELDS stores the definitions of table and view columns.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$FIELD_NAME</td>
<td>CHAR(31)</td>
<td>Column name</td>
</tr>
<tr>
<td>RDB$RELATION_NAME</td>
<td>CHAR(31)</td>
<td>The name of the table or view that the column belongs to</td>
</tr>
<tr>
<td>RDB$FIELD_SOURCE</td>
<td>CHAR(31)</td>
<td>Domain name on which the column is based, either a user-defined one specified in the table definition or one created automatically by the system using the set of attributes defined. The attributes are in the table RDB$FIELDS: this column matches RDB$FIELDS.RDB$FIELD_NAME.</td>
</tr>
<tr>
<td>RDB$QUERY_NAME</td>
<td>CHAR(31)</td>
<td>Not currently used</td>
</tr>
<tr>
<td>RDB$BASE_FIELD</td>
<td>CHAR(31)</td>
<td>Only populated for a view, it is the name of the column from the base table</td>
</tr>
<tr>
<td>RDB$EDIT_STRING</td>
<td>VARCHAR(127)</td>
<td>Not used</td>
</tr>
<tr>
<td>RDB$FIELD_POSITION</td>
<td>SMALLINT</td>
<td>The zero-based ordinal position of the column in the table or view, numbering from left to right</td>
</tr>
<tr>
<td>RDB$QUERY_HEADER</td>
<td>BLOB TEXT</td>
<td>Not used</td>
</tr>
<tr>
<td>RDB$UPDATE_FLAG</td>
<td>SMALLINT</td>
<td>Indicates whether the column is a regular one (value 1) or a computed one (value 0)</td>
</tr>
<tr>
<td>RDB$FIELD_ID</td>
<td>SMALLINT</td>
<td>An ID assigned from RDB$RELATIONS.RDB$FIELD_ID at the time the column was added to the table or view. It should always be treated as transient</td>
</tr>
<tr>
<td>RDB$VIEW_CONTEXT</td>
<td>SMALLINT</td>
<td>For a view column, the internal identifier of the base table from which this field derives</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Comments related to the table or view column</td>
</tr>
<tr>
<td>RDB$DEFAULT_VALUE</td>
<td>BLOB BLR</td>
<td>The value stored for the DEFAULT clause for this column, if there is one, written in binary language representation (BLR)</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Indicates whether the column is user-defined (value 0) or system-defined (value 1 or greater)</td>
</tr>
<tr>
<td>RDB$SECURITY_CLASS</td>
<td>CHAR(31)</td>
<td>May reference a security class defined in RDB$SECURITY_CLASSES, in order to apply access control limits to all users of this column</td>
</tr>
<tr>
<td>RDB$COMPLEX_NAME</td>
<td>CHAR(31)</td>
<td>Not used</td>
</tr>
<tr>
<td>RDB$NULL_FLAG</td>
<td>SMALLINT</td>
<td>Indicates whether the column is nullable (NULL) non-nullable (value 1)</td>
</tr>
<tr>
<td>RDB$DEFAULT_SOURCE</td>
<td>BLOB TEXT</td>
<td>The source text of the DEFAULT clause, if any</td>
</tr>
<tr>
<td>RDB$COLLATION_ID</td>
<td>SMALLINT</td>
<td>The identifier of the collation sequence in the character set for the column, if it is not the default collation</td>
</tr>
<tr>
<td>RDB$GENERATOR_NAME</td>
<td>CHAR(31)</td>
<td>Internal generator name for generating an identity value for the column.</td>
</tr>
<tr>
<td>RDB$IDENTITY_TYPE</td>
<td>SMALLINT</td>
<td>The identity type of the column</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NULL - not an identity column</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - identity column, GENERATED ALWAYS (not supported in Firebird 3.0, will be introduced in Firebird 4.0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - identity column, GENERATED BY DEFAULT</td>
</tr>
</tbody>
</table>

**RDB$ROLES**

RDB$ROLES stores the roles that have been defined in this database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$ROLE_NAME</td>
<td>CHAR(31)</td>
<td>Role name</td>
</tr>
<tr>
<td>RDB$OWNER_NAME</td>
<td>CHAR(31)</td>
<td>The user name of the role owner</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Could store comments related to the role</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>System flag</td>
</tr>
</tbody>
</table>
### RDB$SECURITY_CLASSES

`RDB$SECURITY_CLASSES` stores the access control lists.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$SECURITY_CLASS</td>
<td>CHAR(31)</td>
<td>Security class name</td>
</tr>
<tr>
<td>RDB$ACL</td>
<td>BLOB ACL</td>
<td>The access control list related to the security class. It enumerates users and their privileges</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Could store comments related to the security class</td>
</tr>
</tbody>
</table>

### RDB$TRANSACTIONS

`RDB$TRANSACTIONS` stores the states of distributed transactions and other transactions that were prepared for two-phase commit with an explicit prepare message.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$TRANSACTION_ID</td>
<td>INTEGER</td>
<td>The unique identifier of the transaction being tracked</td>
</tr>
<tr>
<td>RDB$TRANSACTION_STATE</td>
<td>SMALLINT</td>
<td>Transaction state:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - in limbo</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - committed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - rolled back</td>
</tr>
<tr>
<td>RDB$TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>Not used</td>
</tr>
<tr>
<td>RDB$TRANSACTION_DESCRIPTION</td>
<td>BLOB</td>
<td>Describes the prepared transaction and could be a custom message supplied to <code>isc_prepare_transaction2</code>, even if it is not a distributed transaction. It may be used when a lost connection cannot be restored</td>
</tr>
</tbody>
</table>

### RDB$TRIGGERS

`RDB$TRIGGERS` stores the trigger definitions for all tables and views.
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$TRIGGER_NAME</td>
<td>CHAR(31)</td>
<td>Trigger name</td>
</tr>
<tr>
<td>RDB$RELATION_NAME</td>
<td>CHAR(31)</td>
<td>The name of the table or view the trigger applies to. NULL if the trigger is applicable to a database event (&quot;database trigger&quot;)</td>
</tr>
<tr>
<td>RDB$TRIGGER_SEQUENCE</td>
<td>SMALLINT</td>
<td>Position of this trigger in the sequence. Zero usually means that no sequence position is specified</td>
</tr>
<tr>
<td>RDB$TRIGGER_TYPE</td>
<td>BIGINT</td>
<td>The event the trigger fires on, see <code>RDB$TRIGGER_TYPE Value</code></td>
</tr>
<tr>
<td>RDB$TRIGGER_SOURCE</td>
<td>BLOB TEXT</td>
<td>Stores the source code of the trigger in PSQL</td>
</tr>
<tr>
<td>RDB$TRIGGER_BLR</td>
<td>BLOB BLR</td>
<td>Stores the trigger in the binary language representation (BLR)</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Trigger comment text</td>
</tr>
<tr>
<td>RDB$TRIGGER_INACTIVE</td>
<td>SMALLINT</td>
<td>Indicates whether the trigger is currently inactive (1) or active (0)</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Flag: indicates whether the trigger is user-defined (value 0) or system-defined (value 1 or greater)</td>
</tr>
<tr>
<td>RDB$FLAGS</td>
<td>SMALLINT</td>
<td>Internal use</td>
</tr>
<tr>
<td>RDB$VALID_BLR</td>
<td>SMALLINT</td>
<td>Indicates whether the text of the trigger remains valid after the latest modification by the the ALTER TRIGGER statement</td>
</tr>
<tr>
<td>RDB$DEBUG_INFO</td>
<td>BLOB</td>
<td>Contains debugging information about variables used in the trigger</td>
</tr>
<tr>
<td>RDB$ENGINE_NAME</td>
<td>CHAR(31)</td>
<td>Engine for external triggers. 'UDR' for UDR triggers. NULL for PSQL triggers</td>
</tr>
<tr>
<td>RDB$ENTRYPOINT</td>
<td>CHAR(255)</td>
<td>The exported name of the external trigger in the trigger library. Note, this is often not the same as RDB$TRIGGER_NAME, which is the identifier with which the trigger is declared to the database</td>
</tr>
</tbody>
</table>

### RDB$TRIGGER_TYPE Value

The value of `RDB$TRIGGER_TYPE` is built from:

1. **before insert**
Identification of the exact `RDB$TRIGGER_TYPE` code is a little more complicated, since it is a bitmap, calculated according to which phase and events are covered and the order in which they are defined. For the curious, the calculation is explained in this code comment by Mark Rotteveel:

For DDL triggers, the trigger type is obtained by bitwise OR above the event phase (0 — BEFORE, 1 — AFTER) and all listed types events:

- CREATE TABLE: `0x0000000000004002`
- ALTER TABLE: `0x0000000000004004`
- DROP TABLE: `0x0000000000004008`
- CREATE PROCEDURE: `0x0000000000004010`
- ALTER PROCEDURE: `0x0000000000004020`
<table>
<thead>
<tr>
<th>Code</th>
<th>SQL Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000004040</td>
<td>DROP PROCEDURE</td>
</tr>
<tr>
<td>0x0000000000004080</td>
<td>CREATE FUNCTION</td>
</tr>
<tr>
<td>0x0000000000004100</td>
<td>ALTER FUNCTION</td>
</tr>
<tr>
<td>0x0000000000004200</td>
<td>DROP FUNCTION</td>
</tr>
<tr>
<td>0x0000000000004400</td>
<td>CREATE TRIGGER</td>
</tr>
<tr>
<td>0x0000000000004800</td>
<td>ALTER TRIGGER</td>
</tr>
<tr>
<td>0x0000000000005000</td>
<td>DROP TRIGGER</td>
</tr>
<tr>
<td>0x0000000000001400</td>
<td>CREATE EXCEPTION</td>
</tr>
<tr>
<td>0x0000000000002400</td>
<td>ALTER EXCEPTION</td>
</tr>
<tr>
<td>0x0000000000004400</td>
<td>DROP EXCEPTION</td>
</tr>
<tr>
<td>0x0000000000008400</td>
<td>CREATE VIEW</td>
</tr>
<tr>
<td>0x00000000000010400</td>
<td>ALTER VIEW</td>
</tr>
<tr>
<td>0x00000000000020400</td>
<td>DROP VIEW</td>
</tr>
<tr>
<td>0x0000000000004040</td>
<td>CREATE DOMAIN</td>
</tr>
<tr>
<td>0x00000000000088400</td>
<td>ALTER DOMAIN</td>
</tr>
<tr>
<td>0x00000000000100400</td>
<td>DROP DOMAIN</td>
</tr>
<tr>
<td>0x00000000000200400</td>
<td>CREATE ROLE</td>
</tr>
<tr>
<td>0x00000000000400400</td>
<td>ALTER ROLE</td>
</tr>
<tr>
<td>0x00000000000800400</td>
<td>DROP ROLE</td>
</tr>
<tr>
<td>0x00000000000100400</td>
<td>CREATE INDEX</td>
</tr>
<tr>
<td>0x00000000000200400</td>
<td>ALTER INDEX</td>
</tr>
<tr>
<td>0x00000000000400400</td>
<td>DROP INDEX</td>
</tr>
<tr>
<td>0x00000000000800400</td>
<td>CREATE SEQUENCE</td>
</tr>
<tr>
<td>0x00000000001000400</td>
<td>ALTER SEQUENCE</td>
</tr>
<tr>
<td>0x00000000002000400</td>
<td>DROP SEQUENCE</td>
</tr>
<tr>
<td>0x00000000004000400</td>
<td>CREATE USER</td>
</tr>
<tr>
<td>0x00000000008000400</td>
<td>ALTER USER</td>
</tr>
<tr>
<td>0x0000000010000400</td>
<td>DROP USER</td>
</tr>
<tr>
<td>0x0000000020000400</td>
<td>CREATE COLLATION</td>
</tr>
<tr>
<td>0x0000000040000400</td>
<td>DROP COLLATION</td>
</tr>
<tr>
<td>0x0000000050000400</td>
<td>ALTER CHARACTER SET</td>
</tr>
</tbody>
</table>
CREATE PACKAGE
ALTER PACKAGE
DROP PACKAGE
CREATE PACKAGE BODY
DROP PACKAGE BODY
CREATE MAPPING
ALTER MAPPING
DROP MAPPING
ANY DDL STATEMENT

For example a trigger with
BEFORE CREATE PROCEDURE OR CREATE FUNCTION will be of type 0x0000000000004090,
AFTER CREATE PROCEDURE OR CREATE FUNCTION — 0x0000000000004091,
BEFORE DROP FUNCTION OR DROP EXCEPTION — 0x00000000000044200,
AFTER DROP FUNCTION OR DROP EXCEPTION — 0x00000000000044201,
BEFORE DROP TRIGGER OR DROP DOMAIN — 0x00000000001005000,
AFTER DROP TRIGGER OR DROP DOMAIN — 0x00000000001005001.

**RDB$TRIGGER_MESSAGES**

RDB$TRIGGER_MESSAGES stores the trigger messages.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$TRIGGER_NAME</td>
<td>CHAR(31)</td>
<td>The name of the trigger the message is associated with</td>
</tr>
<tr>
<td>RDB$MESSAGE_NUMBER</td>
<td>SMALLINT</td>
<td>The number of the message within this trigger (from 1 to 32,767)</td>
</tr>
<tr>
<td>RDB$MESSAGE</td>
<td>VARCHAR(1023)</td>
<td>Text of the trigger message</td>
</tr>
</tbody>
</table>

**RDB$TYPES**

RDB$TYPES stores the defining sets of enumerated types used throughout the system.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$FIELD_NAME</td>
<td>CHAR(31)</td>
<td>Enumerated type name. Each type name masters its own set of types, e.g., object types, data types, character sets, trigger types, blob subtypes, etc.</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| RDB$TYPE           | SMALLINT  | The object type identifier. A unique series of numbers is used within each separate enumerated type. For example, in this selection from the set mastered under RDB$OBJECT_TYPE in RDB$FIELD_NAME, some object types are enumerated:  
0 - TABLE  
1 - VIEW  
2 - TRIGGER  
3 - COMPUTED_FIELD  
4 - VALIDATION  
5 - PROCEDURE  
... |
| RDB$TYPE_NAME      | CHAR(31)  | The name of a member of an enumerated type, e.g., TABLE, VIEW, TRIGGER, etc. in the example above. In the RDB$CHARACTER_SET enumerated type, RDB$TYPE_NAME stores the names of the character sets. |
| RDB$DESCRIPTION    | BLOB TEXT | Any text comments related to the enumerated type |
| RDB$SYSTEM_FLAG    | SMALLINT  | Flag: indicates whether the type-member is user-defined (value 0) or system-defined (value 1 or greater) |

**RDB$USER_PRIVILEGES**

RDB$USER_PRIVILEGES stores the SQL access privileges for Firebird users and privileged objects.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$USER</td>
<td>CHAR(31)</td>
<td>The user or object that is granted this privilege</td>
</tr>
<tr>
<td>RDB$GRANTOR</td>
<td>CHAR(31)</td>
<td>The user who grants the privilege</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$PRIVILEGE</td>
<td>CHAR(6)</td>
<td>The privilege granted hereby:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A - all (all privileges)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S - select (selecting data)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I - insert (inserting rows)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D - delete (deleting rows)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R - references (foreign key)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U - update (updating data)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X - executing (procedure)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>G - usage (of other object types)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>M - role membership</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C - DDL privilege create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L - DDL privilege alter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - DDL privilege drop</td>
</tr>
<tr>
<td>RDB$GRANT_OPTION</td>
<td>SMALLINT</td>
<td>Whether the WITH GRANT OPTION authority is included with the privilege:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - not included</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - included</td>
</tr>
<tr>
<td>RDB$RELATION_NAME</td>
<td>CHAR(31)</td>
<td>The name of the object (table, view, procedure or role) the privilege is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>granted ON</td>
</tr>
<tr>
<td>RDB$FIELD_NAME</td>
<td>CHAR(31)</td>
<td>The name of the column the privilege is applicable to, for a column-level</td>
</tr>
<tr>
<td></td>
<td></td>
<td>privilege (an UPDATE or REFERENCES privilege)</td>
</tr>
<tr>
<td>RDB$USER_TYPE</td>
<td>SMALLINT</td>
<td>Identifies the type of user the privilege is granted TO (a user, a procedure,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a view, etc.)</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$OBJECT_TYPE</td>
<td>SMALLINT</td>
<td>Identifies the type of the object the privilege is granted ON</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - table</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - view</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 - procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 - exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 - user</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 - domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 - character set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 - role</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14 - generator (sequence)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15 - function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 - BLOB filter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17 - collation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18 - package</td>
</tr>
</tbody>
</table>

RDB$VIEW_RELATIONS

RDB$VIEW_RELATIONS stores the tables that are referred to in view definitions. There is one record for each table in a view.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$VIEW_NAME</td>
<td>CHAR(31)</td>
<td>View name</td>
</tr>
<tr>
<td>RDB$RELATION_NAME</td>
<td>CHAR(31)</td>
<td>The name of the table, view or stored procedure the view references</td>
</tr>
<tr>
<td>RDB$VIEW_CONTEXT</td>
<td>SMALLINT</td>
<td>The alias used to reference the view column in the BLR code of the query definition</td>
</tr>
<tr>
<td>RDB$CONTEXT_NAME</td>
<td>CHAR(255)</td>
<td>The text associated with the alias reported in the RDB$VIEW_CONTEXT column</td>
</tr>
<tr>
<td>RDB$CONTEXT_TYPE</td>
<td>SMALLINT</td>
<td>Context type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - table</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - view</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - stored procedure</td>
</tr>
<tr>
<td>RDB$PACKAGE_NAME</td>
<td>CHAR(31)</td>
<td>Package name for a stored procedure in a package</td>
</tr>
</tbody>
</table>
Appendix E: Monitoring Tables

The Firebird engine can monitor activities in a database and make them available for user queries via the monitoring tables. The definitions of these tables are always present in the database, all named with the prefix MON$. The tables are virtual: they are populated with data only at the moment when the user queries them. That is also one good reason why it is no use trying to create triggers for them!

The key notion in understanding the monitoring feature is an *activity snapshot*. The activity snapshot represents the current state of the database at the start of the transaction in which the monitoring table query runs. It delivers a lot of information about the database itself, active connections, users, transactions prepared, running queries and more.

The snapshot is created when any monitoring table is queried for the first time. It is preserved until the end of the current transaction to maintain a stable, consistent view for queries across multiple tables, such as a master-detail query. In other words, monitoring tables always behave as though they were in SNAPSHOT TABLE STABILITY (“consistency”) isolation, even if the current transaction is started with a lower isolation level.

To refresh the snapshot, the current transaction must be completed and the monitoring tables must be re-queried in a new transaction context.

*Access Security*

- SYSDBA and the database owner have full access to all information available from the monitoring tables
- Regular users can see information about their own connections; other connections are not visible to them

![Warning]

In a highly loaded environment, collecting information via the monitoring tables could have a negative impact on system performance.

*List of Monitoring Tables*

**MON$ATTACHMENTS**

Information about active attachments to the database

**MON$CALL_STACK**

Calls to the stack by active queries of stored procedures and triggers

**MON$CONTEXT_VARIABLES**

Information about custom context variables

**MON$DATABASE**

Information about the database to which the CURRENT_CONNECTION is attached

**MON$IO_STATS**

Input/output statistics
**MON$ATTACHMENTS**

**MON$ATTACHMENTS** displays information about active attachments to the database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MON$ATTACHMENT_ID</td>
<td>BIGINT</td>
<td>Connection identifier</td>
</tr>
<tr>
<td>MON$SERVER_PID</td>
<td>INTEGER</td>
<td>Server process identifier</td>
</tr>
<tr>
<td>MON$STATE</td>
<td>SMALLINT</td>
<td>Connection state:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - idle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - active</td>
</tr>
<tr>
<td>MON$ATTACHMENT_NAME</td>
<td>VARCHAR(255)</td>
<td>Connection string — the file name and full path to the primary database file</td>
</tr>
<tr>
<td>MON$USER</td>
<td>CHAR(31)</td>
<td>The name of the user who is using this connection</td>
</tr>
<tr>
<td>MON$ROLE</td>
<td>CHAR(31)</td>
<td>The role name specified when the connection was established. If no role</td>
</tr>
<tr>
<td></td>
<td></td>
<td>was specified when the connection was established, the field contains the text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NONE</td>
</tr>
<tr>
<td>MON$REMOTE_PROTOCOL</td>
<td>VARCHAR(10)</td>
<td>Remote protocol name</td>
</tr>
<tr>
<td>MON$REMOTE_ADDRESS</td>
<td>VARCHAR(255)</td>
<td>Remote address (address and server name)</td>
</tr>
<tr>
<td>MON$REMOTE_PID</td>
<td>INTEGER</td>
<td>Remote client process identifier</td>
</tr>
<tr>
<td>MON$CHARACTER_SET_ID</td>
<td>SMALLINT</td>
<td>Connection character set identifier (see RDB$CHARACTER_SET in system table</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RDB$TYPES)</td>
</tr>
<tr>
<td>MON$TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>The date and time when the connection was started</td>
</tr>
</tbody>
</table>
### Column Name | Data Type | Description
--- | --- | ---
MON$GARBAGE_COLLECTION | SMALLINT | Garbage collection flag (as specified in the attachment’s DPB): 1=allowed, 0=not allowed
MON$REMOTE_PROCESS | VARCHAR(255) | The full file name and path to the executable file that established this connection
MON$STAT_ID | INTEGER | Statistics identifier
MON$CLIENT_VERSION | VARCHAR(255) | Client library version
MON$REMOTE_VERSION | VARCHAR(255) | Remote protocol version
MON$REMOTE_HOST | VARCHAR(255) | Name of the remote host
MON$REMOTE_OS_USER | VARCHAR(255) | Name of remote user
MON$AUTH_METHOD | VARCHAR(255) | Name of authentication plugin used to connect
MON$SYSTEM_FLAG | SMALLINT | Flag that indicates the type of connection:
| | | 0 - normal connection
| | | 1 - system connection

#### Retrieving information about client applications

```
SELECT MON$USER, MON$REMOTE_ADDRESS, MON$REMOTE_PID, MON$TIMESTAMP
FROM MON$ATTACHMENTS
WHERE MON$ATTACHMENT_ID <> CURRENT_CONNECTION
```

#### Using MON$ATTACHMENTS to Kill a Connection

Monitoring tables are read-only. However, the server has a built-in mechanism for deleting (and only deleting) records in the MON$ATTACHMENTS table, which makes it possible to close a connection to the database.

### Notes

- All the current activity in the connection being deleted is immediately stopped and all active transactions are rolled back
- The closed connection will return an error with the `isc_att_shutdown` code to the application
- Subsequent attempts to use this connection (i.e., use its handle in API calls) will return errors
- Termination of system connections (`MON$SYSTEM_FLAG = 1`) is not possible. The server will skip system connections in a `DELETE FROM MON$ATTACHMENTS`.
MON$CALL_STACK displays calls to the stack from queries executing in stored procedures and triggers.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MON$CALL_ID</td>
<td>BIGINT</td>
<td>Call identifier</td>
</tr>
<tr>
<td>MON$STATEMENT_ID</td>
<td>BIGINT</td>
<td>The identifier of the top-level SQL statement, the one that initiated the chain of calls. Use this identifier to find the records about the active statement in the MON$STATEMENTS table</td>
</tr>
<tr>
<td>MON$CALLER_ID</td>
<td>BIGINT</td>
<td>The identifier of the calling trigger or stored procedure</td>
</tr>
<tr>
<td>MON$OBJECT_NAME</td>
<td>CHAR(31)</td>
<td>PSQL object (module) name</td>
</tr>
<tr>
<td>MON$OBJECT_TYPE</td>
<td>SMALLINT</td>
<td>PSQL object type (trigger or stored procedure):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 - stored procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15 - stored function</td>
</tr>
<tr>
<td>MON$TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>The date and time when the call was started</td>
</tr>
<tr>
<td>MON$SOURCE_LINE</td>
<td>INTEGER</td>
<td>The number of the source line in the SQL statement being executed at the moment of the snapshot</td>
</tr>
<tr>
<td>MON$SOURCE_COLUMN</td>
<td>INTEGER</td>
<td>The number of the source column in the SQL statement being executed at the moment of the snapshot</td>
</tr>
<tr>
<td>MON$STAT_ID</td>
<td>INTEGER</td>
<td>Statistics identifier</td>
</tr>
<tr>
<td>MON$PACKAGE_NAME</td>
<td>CHAR(31)</td>
<td>Package name for stored procedures or functions in a package</td>
</tr>
</tbody>
</table>

Information about calls during the execution of the EXECUTE STATEMENT statement does not get into the call stack.

Get the call stack for all connections except your own

WITH RECURSIVE
HEAD AS ( 
MON$CONTEXT_VARIABLES

MON$CONTEXT_VARIABLES displays information about custom context variables.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MON$ATTACHMENT_ID</td>
<td>BIGINT</td>
<td>Connection identifier. It contains a valid value only for a connection-level context variable. For transaction-level variables it is NULL.</td>
</tr>
<tr>
<td>MON$TRANSACTION_ID</td>
<td>BIGINT</td>
<td>Transaction identifier. It contains a valid value only for transaction-level context variables. For connection-level variables it is NULL.</td>
</tr>
<tr>
<td>MON$VARIABLE_NAME</td>
<td>VARCHAR(80)</td>
<td>Context variable name</td>
</tr>
<tr>
<td>MON$VARIABLE_VALUE</td>
<td>VARCHAR(32765)</td>
<td>Context variable value</td>
</tr>
</tbody>
</table>

Retrieving all session context variables for the current connection

```sql
SELECT
    VAR.MON$VARIABLE_NAME,
    VAR.MON$VARIABLE_VALUE
FROM MON$CONTEXT_VARIABLES VAR
WHERE VAR.MON$ATTACHMENT_ID = CURRENT_CONNECTION
```

MON$DATABASE

MON$DATABASE displays the header information from the database the current user is connected to.
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MON$DATABASE_NAME</td>
<td>VARCHAR(255)</td>
<td>The file name and full path of the primary database file, or the database alias</td>
</tr>
<tr>
<td>MON$PAGE_SIZE</td>
<td>SMALLINT</td>
<td>Database page size in bytes</td>
</tr>
<tr>
<td>MON$ODS_MAJOR</td>
<td>SMALLINT</td>
<td>Major ODS version, e.g., 11</td>
</tr>
<tr>
<td>MON$ODS_MINOR</td>
<td>SMALLINT</td>
<td>Minor ODS version, e.g., 2</td>
</tr>
<tr>
<td>MON$OLDEST_TRANSACTION</td>
<td>BIGINT</td>
<td>The number of the oldest [interesting] transaction (OIT)</td>
</tr>
<tr>
<td>MON$OLDEST_ACTIVE</td>
<td>BIGINT</td>
<td>The number of the oldest active transaction (OAT)</td>
</tr>
<tr>
<td>MON$OLDEST_SNAPSHOT</td>
<td>BIGINT</td>
<td>The number of the transaction that was active at the moment when the OAT was started — oldest snapshot transaction (OST)</td>
</tr>
<tr>
<td>MON$NEXT_TRANSACTION</td>
<td>BIGINT</td>
<td>The number of the next transaction, as it stood when the monitoring snapshot was taken</td>
</tr>
<tr>
<td>MON$PAGE_BUFFERS</td>
<td>INTEGER</td>
<td>The number of pages allocated in RAM for the database page cache</td>
</tr>
<tr>
<td>MON$SQL_DIALECT</td>
<td>SMALLINT</td>
<td>Database SQL Dialect: 1 or 3</td>
</tr>
<tr>
<td>MON$SHUTDOWN_MODE</td>
<td>SMALLINT</td>
<td>The current shutdown state of the database:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - the database is online</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - multi-user shutdown</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - single-user shutdown</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - full shutdown</td>
</tr>
<tr>
<td>MON$SWEEP_INTERVAL</td>
<td>INTEGER</td>
<td>Sweep interval</td>
</tr>
<tr>
<td>MON$READ_ONLY</td>
<td>SMALLINT</td>
<td>Flag indicating whether the database is read-only (value 1) or read-write (value 0)</td>
</tr>
<tr>
<td>MON$FORCED_WRITES</td>
<td>SMALLINT</td>
<td>Indicates whether the write mode of the database is set for synchronous write (forced writes ON, value is 1) or asynchronous write (forced writes OFF, value is 0)</td>
</tr>
<tr>
<td>MON$RESERVE_SPACE</td>
<td>SMALLINT</td>
<td>The flag indicating reserve_space (value 1) or use_all_space (value 0) for filling database pages</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MON$CREATION_DATE</td>
<td>TIMESTAMP</td>
<td>The date and time when the database was created or was last restored</td>
</tr>
<tr>
<td>MON$PAGES</td>
<td>BIGINT</td>
<td>The number of pages allocated for the database on an external device</td>
</tr>
<tr>
<td>MON$STAT_ID</td>
<td>INTEGER</td>
<td>Statistics identifier</td>
</tr>
<tr>
<td>MON$BACKUP_STATE</td>
<td>SMALLINT</td>
<td>Current physical backup (nBackup) state:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - normal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - stalled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - merge</td>
</tr>
<tr>
<td>MON$CRYPT_PAGE</td>
<td>BIGINT</td>
<td>Number of encrypted pages</td>
</tr>
<tr>
<td>MON$OWNER</td>
<td>CHAR(31)</td>
<td>Username of the database owner</td>
</tr>
<tr>
<td>MON$SEC_DATABASE</td>
<td>CHAR(7)</td>
<td>Displays what type of security database is used:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default - default security database, i.e. security3.fdb</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Self - current database is used as security database</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other - another database is used as security database (not itself or security3.fdb)</td>
</tr>
</tbody>
</table>

### MON$IO_STATS

**MON$IO_STATS** displays input/output statistics. The counters are cumulative, by group, for each group of statistics.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MON$STAT_ID</td>
<td>INTEGER</td>
<td>Statistics identifier</td>
</tr>
<tr>
<td>MON$STAT_GROUP</td>
<td>SMALLINT</td>
<td>Statistics group:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - database</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - connection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - transaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - statement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 - call</td>
</tr>
<tr>
<td>MON$PAGE_READS</td>
<td>BIGINT</td>
<td>Count of database pages read</td>
</tr>
<tr>
<td>MON$PAGE_WRITES</td>
<td>BIGINT</td>
<td>Count of database pages written to</td>
</tr>
<tr>
<td>MON$PAGE_FETCHES</td>
<td>BIGINT</td>
<td>Count of database pages fetched</td>
</tr>
</tbody>
</table>
### MON$MEMORY_USAGE

**MON$MEMORY_USAGE** displays memory usage statistics.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MON$STAT_ID</td>
<td>INTEGER</td>
<td>Statistics identifier</td>
</tr>
<tr>
<td>MON$STAT_GROUP</td>
<td>SMALLINT</td>
<td>Statistics group: 0 - database, 1 - connection, 2 - transaction, 3 - operator, 4 - call</td>
</tr>
<tr>
<td>MON$MEMORY_USED</td>
<td>BIGINT</td>
<td>The amount of memory in use, in bytes.  This data is about the high-level memory allocation performed by the server. It can be useful to track down memory leaks and excessive memory usage in connections, procedures, etc.</td>
</tr>
<tr>
<td>MON$MEMORY_ALLOCATED</td>
<td>BIGINT</td>
<td>The amount of memory allocated by the operating system, in bytes. This data is about the low-level memory allocation performed by the Firebird memory manager — the amount of memory allocated by the operating system — which can allow you to control the physical memory usage.</td>
</tr>
<tr>
<td>MON$MAX_MEMORY_USED</td>
<td>BIGINT</td>
<td>The maximum number of bytes used by this object</td>
</tr>
<tr>
<td>MON$MAX_MEMORY_ALLOCATED</td>
<td>BIGINT</td>
<td>The maximum number of bytes allocated for this object by the operating system</td>
</tr>
</tbody>
</table>

Counters associated with database-level records **MON$DATABASE (MON$STAT_GROUP = 0)**, display memory allocation for all connections. In the Classic and SuperClassic zero values of the counters indicate that these architectures have no common cache.

Minor memory allocations are not accrued here but are added to the database memory pool instead.
Getting 10 requests consuming the most memory

```sql
SELECT
    STMT.MON$ATTACHMENT_ID,
    STMT.MON$SQL_TEXT,
    MEM.MON$MEMORY_USED
FROM MON$MEMORY_USAGE MEM
NATURAL JOIN MON$STATEMENTS STMT
ORDER BY MEM.MON$MEMORY_USED DESC
FETCH FIRST 10 ROWS ONLY
```

**MON$RECORD_STATS**

`MON$RECORD_STATS` displays record-level statistics. The counters are cumulative, by group, for each group of statistics.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MON$STAT_ID</td>
<td>INTEGER</td>
<td>Statistics identifier</td>
</tr>
<tr>
<td>MON$STAT_GROUP</td>
<td>SMALLINT</td>
<td>Statistics group:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - database</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - connection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - transaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - statement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 - call</td>
</tr>
<tr>
<td>MON$RECORD_SEQ_READS</td>
<td>BIGINT</td>
<td>Count of records read sequentially</td>
</tr>
<tr>
<td>MON$RECORD_IDX_READS</td>
<td>BIGINT</td>
<td>Count of records read via an index</td>
</tr>
<tr>
<td>MON$RECORD_INSERTS</td>
<td>BIGINT</td>
<td>Count of inserted records</td>
</tr>
<tr>
<td>MON$RECORD_UPDATES</td>
<td>BIGINT</td>
<td>Count of updated records</td>
</tr>
<tr>
<td>MON$RECORD_DELETES</td>
<td>BIGINT</td>
<td>Count of deleted records</td>
</tr>
<tr>
<td>MON$RECORD_BACKOUTS</td>
<td>BIGINT</td>
<td>Count of records backed out</td>
</tr>
<tr>
<td>MON$RECORD_PURGES</td>
<td>BIGINT</td>
<td>Count of records purged</td>
</tr>
<tr>
<td>MON$RECORD_EXPUNGES</td>
<td>BIGINT</td>
<td>Count of records expunged</td>
</tr>
<tr>
<td>MON$RECORD_LOCKS</td>
<td>BIGINT</td>
<td>Number of records locked</td>
</tr>
<tr>
<td>MON$RECORD_WAITS</td>
<td>BIGINT</td>
<td>Number of update, delete or lock attempts on records owned by other active transactions. Transaction is in WAIT mode.</td>
</tr>
<tr>
<td>MON$RECORD_CONFLICTS</td>
<td>BIGINT</td>
<td>Number of unsuccessful update, delete or lock attempts on records owned by other active transactions. These are reported as update conflicts.</td>
</tr>
</tbody>
</table>
**MON$STATEMENTS**

MON$STATEMENTS displays statements prepared for execution.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MON$STATEMENT_ID</td>
<td>BIGINT</td>
<td>Statement identifier</td>
</tr>
<tr>
<td>MON$ATTACHMENT_ID</td>
<td>BIGINT</td>
<td>Connection identifier</td>
</tr>
<tr>
<td>MON$TRANSACTION_ID</td>
<td>BIGINT</td>
<td>Transaction identifier</td>
</tr>
<tr>
<td>MON$STATE</td>
<td>SMALLINT</td>
<td>Statement state:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - idle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - active</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - stalled</td>
</tr>
<tr>
<td>MON$TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>The date and time when the statement was prepared</td>
</tr>
<tr>
<td>MON$SQL_TEXT</td>
<td>BLOB TEXT</td>
<td>Statement text in SQL</td>
</tr>
<tr>
<td>MON$STAT_ID</td>
<td>INTEGER</td>
<td>Statistics identifier</td>
</tr>
<tr>
<td>MON$EXPLAINED_PLAN</td>
<td>BLOB TEXT</td>
<td>Explained execution plan</td>
</tr>
</tbody>
</table>

The STALLED state indicates that, at the time of the snapshot, the statement had an open cursor and was waiting for the client to resume fetching rows.

*Display active queries, excluding those running in your connection*

```
SELECT ATT.MON$USER,
       ATT.MON$REMOTE_ADDRESS,
       STMT.MON$SQL_TEXT,
       STMT.MON$TIMESTAMP
FROM MON$ATTACHMENTS ATT
JOIN MON$STATEMENTS STMT ON ATT.MON$ATTACHMENT_ID = STMT.MON$ATTACHMENT_ID
WHERE ATT.MON$ATTACHMENT_ID <> CURRENT_CONNECTION
     AND STMT.MON$STATE = 1
```

**Using MON$STATEMENTS to Cancel a Query**

Monitoring tables are read-only. However, the server has a built-in mechanism for deleting (and only deleting) records in the MON$STATEMENTS table, which makes it possible to cancel a running
If no statements are currently being executed in the connection, any attempt to cancel queries will not proceed.

After a query is cancelled, calling execute/fetch API functions will return an error with the `isc_cancelled` code.

Subsequent queries from this connection will proceed as normal.

Cancellation of the statement does not occur synchronously, it only marks the request for cancellation, and the cancellation itself is done asynchronously by the server.

Example

Cancelling all active queries for the specified connection:

```sql
DELETE FROM MON$STATEMENTS
WHERE MON$ATTACHMENT_ID = 32
```

**MON$TABLE_STATS**

MON$TABLE_STATS reports table-level statistics.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MON$STAT_ID</td>
<td>INTEGER</td>
<td>Statistics identifier</td>
</tr>
<tr>
<td>MON$STAT_GROUP</td>
<td>SMALLINT</td>
<td>Statistics group:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - database</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - connection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - transaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - statement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 - call</td>
</tr>
<tr>
<td>MON$TABLE_NAME</td>
<td>CHAR(31)</td>
<td>Name of the table</td>
</tr>
<tr>
<td>MON$RECORD_STAT_ID</td>
<td>INTEGER</td>
<td>Link to MON$RECORD_STATS</td>
</tr>
</tbody>
</table>

Getting statistics at the record level for each table for the current connection:

```sql
SELECT
  t.mon$table_name,
  r.mon$record_inserts,
  r.mon$record_updates,
  r.mon$record_deletes,
  r.mon$record_backouts,
  r.mon$record_purges,
  r.mon$record_expunges,
```
MON$TRANSACTIONS

MON$TRANSACTIONS reports started transactions.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MON$TRANSACTION_ID</td>
<td>BIGINT</td>
<td>Transaction identifier (number)</td>
</tr>
<tr>
<td>MON$ATTACHMENT_ID</td>
<td>BIGINT</td>
<td>Connection identifier</td>
</tr>
<tr>
<td>MON$STATE</td>
<td>SMALLINT</td>
<td>Transaction state:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - idle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - active</td>
</tr>
<tr>
<td>MON$TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>The date and time when the transaction was started</td>
</tr>
<tr>
<td>MON$TOP_TRANSACTION</td>
<td>BIGINT</td>
<td>Top-level transaction identifier (number)</td>
</tr>
<tr>
<td>MON$OLDEST_TRANSACTION</td>
<td>BIGINT</td>
<td>Transaction ID of the oldest [interesting] transaction (OIT)</td>
</tr>
<tr>
<td>MON$OLDEST_ACTIVE</td>
<td>BIGINT</td>
<td>Transaction ID of the oldest active transaction (OAT)</td>
</tr>
<tr>
<td>MON$ISOLATION_MODE</td>
<td>SMALLINT</td>
<td>Isolation mode (level):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - consistency (snapshot table stability)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - concurrency (snapshot)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - read committed record version</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - read committed no record version</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MON$LOCK_TIMEOUT</td>
<td>SMALLINT</td>
<td>Lock timeout:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1 - wait forever</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - no waiting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 or greater - lock timeout in seconds</td>
</tr>
<tr>
<td>MON$READ_ONLY</td>
<td>SMALLINT</td>
<td>Flag indicating whether the transaction is read-only (value 1) or read-write (value 0)</td>
</tr>
<tr>
<td>MON$AUTO_COMMIT</td>
<td>SMALLINT</td>
<td>Flag indicating whether automatic commit is used for the transaction (value 1) or not (value 0)</td>
</tr>
<tr>
<td>MON$AUTO_UNDO</td>
<td>SMALLINT</td>
<td>Flag indicating whether the logging mechanism automatic undo is used for the transaction (value 1) or not (value 0)</td>
</tr>
<tr>
<td>MON$STAT_ID</td>
<td>INTEGER</td>
<td>Statistics identifier</td>
</tr>
</tbody>
</table>

*Getting all connections that started Read Write transactions with isolation level above Read Committed*

```sql
SELECT DISTINCT a.*
FROM mon$attachments a
JOIN mon$transactions t ON a.mon$attachment_id = t.mon$attachment_id
WHERE NOT (t.mon$read_only = 1 AND t.mon$isolation_mode >= 2)
```
Appendix F: Security tables

The names of the security tables have SEC$ as prefix. They display data from the current security database. These tables are virtual in the sense that before access by the user, no data is recorded in them. They are filled with data at the time of user request. However, the descriptions of these tables are constantly present in the database. In this sense, these virtual tables are like tables of the MON$-family used to monitor the server.

Security

- SYSDBA, users with the RDB$ADMIN role in the security database and the current database, and the owner of the security database have full access to all information provided by the security tables.
- Regular users can only see information on themselves, other users are not visible.

These features are highly dependent on the user management plugin. Keep in mind that some options are ignored when using a legacy control plugin users.

List of security tables

**SEC$DB_CREATORS**

Lists users and roles granted the CREATE DATABASE privilege

**SEC$GLOBAL_AUTH_MAPPING**

Information about global authentication mappings

**SEC$USERS**

Lists users in the current security database

**SEC$USER_ATTRIBUTES**

Additional attributes of users

**SEC$DB_CREATORS**

Lists users and roles granted the CREATE DATABASE privilege.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEC$USER</td>
<td>CHAR(31)</td>
<td>Name of the user or role</td>
</tr>
<tr>
<td>SEC$USER_TYPE</td>
<td>SMALLINT</td>
<td>Type of user:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 - user</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 - role</td>
</tr>
</tbody>
</table>

**SEC$GLOBAL_AUTH_MAPPING**

Lists users and roles granted the CREATE DATABASE privilege.
### Column Name | Data Type | Description
--- | --- | ---
SEC$MAP_NAME | CHAR(31) | Name of the mapping
SEC$MAP_USING | CHAR(1) | Using definition:
P - plugin (specific or any)
S - any plugin serverwide
M - mapping
* - any method
SEC$MAP_PLUGIN | CHAR(31) | Mapping applies for authentication information from this specific plugin
SEC$MAP_DB | CHAR(31) | Mapping applies for authentication information from this specific database
SEC$MAP_FROM_TYPE | CHAR(31) | The type of authentication object (defined by plugin) to map from, or * for any type
SEC$MAP_FROM | CHAR(255) | The name of the authentication object to map from
SEC$MAP_TO_TYPE | SMALLINT Nullable | The type to map to
0 - USER
1 - ROLE
SEC$MAP_TO | CHAR(31) | The name to map to

### SEC$USERS
Lists users in the current security database.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEC$USER_NAME</td>
<td>CHAR(31)</td>
<td>Username</td>
</tr>
<tr>
<td>SEC$FIRST_NAME</td>
<td>VARCHAR(32)</td>
<td>First name</td>
</tr>
<tr>
<td>SEC$MIDDLE_NAME</td>
<td>VARCHAR(32)</td>
<td>Middle name</td>
</tr>
<tr>
<td>SEC$LAST_NAME</td>
<td>VARCHAR(32)</td>
<td>Last name</td>
</tr>
<tr>
<td>SEC$ACTIVE</td>
<td>BOOLEAN</td>
<td>true - active, false - inactive</td>
</tr>
<tr>
<td>SEC$ADMIN</td>
<td>BOOLEAN</td>
<td>true - user has admin role in security database, false otherwise</td>
</tr>
<tr>
<td>SEC$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Description (comment) on the user</td>
</tr>
<tr>
<td>SEC$PLUGIN</td>
<td>CHAR(31)</td>
<td>Authentication plugin name that manages this user</td>
</tr>
</tbody>
</table>

⚠️ Multiple users can exist with the same username, each managed by a different authentication plugin.
### SEC$USER_ATTRIBUTES

Additional attributes of users

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEC$USER_NAME</td>
<td>CHAR(31)</td>
<td>Username</td>
</tr>
<tr>
<td>SEC$KEY</td>
<td>VARCHAR(31)</td>
<td>Attribute name</td>
</tr>
<tr>
<td>SEC$VALUE</td>
<td>VARCHAR(255)</td>
<td>Attribute value</td>
</tr>
<tr>
<td>SEC$PLUGIN</td>
<td>CHAR(31)</td>
<td>Authentication plugin name that manages this user</td>
</tr>
</tbody>
</table>

Displaying a list of users and their attributes

```sql
SELECT
    U.SEC$USER_NAME AS LOGIN,
    A.SEC$KEY AS TAG,
    A.SEC$VALUE AS "VALUE",
    U.SEC$PLUGIN AS "PLUGIN"
FROM SEC$USERS U
LEFT JOIN SEC$USER_ATTRIBUTES A
    ON U.SEC$USER_NAME = A.SEC$USER_NAME
    AND U.SEC$PLUGIN = A.SEC$PLUGIN;
```

<table>
<thead>
<tr>
<th>LOGIN</th>
<th>TAG</th>
<th>VALUE</th>
<th>PLUGIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSDBA</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
<td>Srp</td>
</tr>
<tr>
<td>ALEX</td>
<td>B</td>
<td>x</td>
<td>Srp</td>
</tr>
<tr>
<td>ALEX</td>
<td>C</td>
<td>sample</td>
<td>Legacy_UserManager</td>
</tr>
</tbody>
</table>
### Appendix G: Character Sets and Collation Sequences

Table 234. Character Sets and Collation Sequences

<table>
<thead>
<tr>
<th>Character Set</th>
<th>ID</th>
<th>Bytes per Char</th>
<th>Collation</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>2</td>
<td>1</td>
<td>ASCII</td>
<td>English</td>
</tr>
<tr>
<td>BIG_5</td>
<td>56</td>
<td>2</td>
<td>BIG_5</td>
<td>Chinese, Vietnamese, Korean</td>
</tr>
<tr>
<td>CP943C</td>
<td>68</td>
<td>2</td>
<td>CP943C</td>
<td>Japanese</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CP943C_UNICODE</td>
<td>Japanese</td>
</tr>
<tr>
<td>CYRL</td>
<td>50</td>
<td>1</td>
<td>CYRL</td>
<td>Russian</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_RUS</td>
<td>Russian dBase</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PDOX_CYRL</td>
<td>Russian Paradox</td>
</tr>
<tr>
<td>DOS437</td>
<td>10</td>
<td>1</td>
<td>DOS437</td>
<td>U.S. English</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_DEU437</td>
<td>German dBase</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_ESP437</td>
<td>Spanish dBase</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_FIN437</td>
<td>Finnish dBase</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_FRA437</td>
<td>French dBase</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_ITA437</td>
<td>Italian dBase</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_NLD437</td>
<td>Dutch dBase</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_SVE437</td>
<td>Swedish dBase</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_UK437</td>
<td>English (Great Britain) dBase</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_US437</td>
<td>U.S. English dBase</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PDOX_ASCII</td>
<td>Code page Paradox-ASCII</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PDOX_INTL</td>
<td>International English Paradox</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PDOX_SWEDFIN</td>
<td>Swedish / Finnish Paradox</td>
</tr>
<tr>
<td>DOS737</td>
<td>9</td>
<td>1</td>
<td>DOS737</td>
<td>Greek</td>
</tr>
<tr>
<td>DOS775</td>
<td>15</td>
<td>1</td>
<td>DOS775</td>
<td>Baltic</td>
</tr>
<tr>
<td>DOS850</td>
<td>11</td>
<td>1</td>
<td>DOS850</td>
<td>Latin I (no Euro symbol)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_DEU850</td>
<td>German</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_ESP850</td>
<td>Spanish</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_FRA850</td>
<td>French</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_FRC850</td>
<td>French-Canada</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DB_ITA850</td>
<td>Italian</td>
</tr>
<tr>
<td>Character Set</td>
<td>ID</td>
<td>Bytes per Char</td>
<td>Collation</td>
<td>Language</td>
</tr>
<tr>
<td>--------------</td>
<td>-----</td>
<td>----------------</td>
<td>-----------</td>
<td>------------------------</td>
</tr>
<tr>
<td>DB_NLD850</td>
<td></td>
<td></td>
<td>DB_NLD850</td>
<td>Dutch</td>
</tr>
<tr>
<td>DB_PTB850</td>
<td></td>
<td></td>
<td>DB_PTB850</td>
<td>Portuguese-Brazil</td>
</tr>
<tr>
<td>DB_SVE850</td>
<td></td>
<td></td>
<td>DB_SVE850</td>
<td>Swedish</td>
</tr>
<tr>
<td>DB_UK850</td>
<td></td>
<td></td>
<td>DB_UK850</td>
<td>English-Great Britain</td>
</tr>
<tr>
<td>DB_US850</td>
<td></td>
<td></td>
<td>DB_US850</td>
<td>U.S. English</td>
</tr>
<tr>
<td>DOS852</td>
<td>45</td>
<td>1</td>
<td>DOS852</td>
<td>Latin II</td>
</tr>
<tr>
<td>DB_CSY</td>
<td></td>
<td></td>
<td>DB_CSY</td>
<td>Czech dBase</td>
</tr>
<tr>
<td>DB_PLK</td>
<td></td>
<td></td>
<td>DB_PLK</td>
<td>Polish dBase</td>
</tr>
<tr>
<td>DB_SLO</td>
<td></td>
<td></td>
<td>DB_SLO</td>
<td>Slovene dBase</td>
</tr>
<tr>
<td>PDOX_CSY</td>
<td></td>
<td></td>
<td>PDOX_CSY</td>
<td>Czech Paradox</td>
</tr>
<tr>
<td>PDOX_HUN</td>
<td></td>
<td></td>
<td>PDOX_HUN</td>
<td>Hungarian Paradox</td>
</tr>
<tr>
<td>PDOX_PLK</td>
<td></td>
<td></td>
<td>PDOX_PLK</td>
<td>Polish Paradox</td>
</tr>
<tr>
<td>PDOX_SLO</td>
<td></td>
<td></td>
<td>PDOX_SLO</td>
<td>Slovene Paradox</td>
</tr>
<tr>
<td>DOS857</td>
<td>46</td>
<td>1</td>
<td>DOS857</td>
<td>Turkish</td>
</tr>
<tr>
<td>DB_TRK</td>
<td></td>
<td></td>
<td>DB_TRK</td>
<td>Turkish dBase</td>
</tr>
<tr>
<td>DOS858</td>
<td>16</td>
<td>1</td>
<td>DOS858</td>
<td>Latin I (with Euro symbol)</td>
</tr>
<tr>
<td>DOS860</td>
<td>13</td>
<td>1</td>
<td>DOS860</td>
<td>Portuguese</td>
</tr>
<tr>
<td>DB_PTG860</td>
<td></td>
<td></td>
<td>DB_PTG860</td>
<td>Portuguese dBase</td>
</tr>
<tr>
<td>DOS861</td>
<td>47</td>
<td>1</td>
<td>DOS861</td>
<td>Icelandic</td>
</tr>
<tr>
<td>PDOX_ISL</td>
<td></td>
<td></td>
<td>PDOX_ISL</td>
<td>Icelandic Paradox</td>
</tr>
<tr>
<td>DOS862</td>
<td>17</td>
<td>1</td>
<td>DOS862</td>
<td>Hebrew</td>
</tr>
<tr>
<td>DOS863</td>
<td>14</td>
<td>1</td>
<td>DOS863</td>
<td>French-Canada</td>
</tr>
<tr>
<td>DB_FRC863</td>
<td></td>
<td></td>
<td>DB_FRC863</td>
<td>French dBase-Canada</td>
</tr>
<tr>
<td>DOS864</td>
<td>18</td>
<td>1</td>
<td>DOS864</td>
<td>Arabic</td>
</tr>
<tr>
<td>DOS865</td>
<td>12</td>
<td>1</td>
<td>DOS865</td>
<td>Scandinavian</td>
</tr>
<tr>
<td>DB_DAN865</td>
<td></td>
<td></td>
<td>DB_DAN865</td>
<td>Danish dBase</td>
</tr>
<tr>
<td>DB_NOR865</td>
<td></td>
<td></td>
<td>DB_NOR865</td>
<td>Norwegian dBase</td>
</tr>
<tr>
<td>PDOX_NORDAN4</td>
<td></td>
<td></td>
<td>PDOX_NORDAN4</td>
<td>Paradox Norway and Denmark</td>
</tr>
<tr>
<td>DOS866</td>
<td>48</td>
<td>1</td>
<td>DOS866</td>
<td>Russian</td>
</tr>
<tr>
<td>DOS869</td>
<td>49</td>
<td>1</td>
<td>DOS869</td>
<td>Modern Greek</td>
</tr>
<tr>
<td>EUCJ_0208</td>
<td>6</td>
<td>2</td>
<td>EUCJ_0208</td>
<td>Japanese EUC</td>
</tr>
<tr>
<td>GB18030</td>
<td>69</td>
<td>4</td>
<td>GB18030</td>
<td>Chinese</td>
</tr>
</tbody>
</table>
## Appendix G: Character Sets and Collation Sequences

<table>
<thead>
<tr>
<th>Character Set</th>
<th>ID</th>
<th>Bytes per Char</th>
<th>Collation</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>GB18030_UNICODE</td>
<td>Chinese</td>
</tr>
<tr>
<td>GBK</td>
<td>67</td>
<td>2</td>
<td>GBK</td>
<td>Chinese</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>GBK_UNICODE</td>
<td>Chinese</td>
</tr>
<tr>
<td>GB_2312</td>
<td>57</td>
<td>2</td>
<td>GB_2312</td>
<td>Simplified Chinese (Hong Kong, Korea)</td>
</tr>
<tr>
<td>ISO8859_1</td>
<td>21</td>
<td>1</td>
<td>ISO8859_1</td>
<td>Latin I</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>DA_DA</td>
<td>Danish</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>DE_DE</td>
<td>German</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>DU_NL</td>
<td>Dutch</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>EN_UK</td>
<td>English-Great Britain</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>EN_US</td>
<td>U.S. English</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>ES_ES</td>
<td>Spanish</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>ES_ES_CI_AI</td>
<td>Spanish — case insensitive and + accent-insensitive</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>FI_FI</td>
<td>Finnish</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>FR_CA</td>
<td>French-Canada</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>FR_CA_CI_AI</td>
<td>French-Canada — case insensitive + accent insensitive</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>FR_FR</td>
<td>French</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>FR_FR_CI_AI</td>
<td>French — case insensitive + accent insensitive</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>IS_IS</td>
<td>Icelandic</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>IT_IT</td>
<td>Italian</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>NO_NO</td>
<td>Norwegian</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PT_BR</td>
<td>Portuguese-Brazil</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PT_PT</td>
<td>Portuguese</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>SV_SV</td>
<td>Swedish</td>
</tr>
<tr>
<td>ISO8859_2</td>
<td>22</td>
<td>1</td>
<td>ISO8859_2</td>
<td>Latin 2 — Central Europe (Croatian, Czech, Hungarian, Polish, Romanian, Serbian, Slovak, Slovenian)</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>CS_CZ</td>
<td>Czech</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>ISO_HUN</td>
<td>Hungarian — case insensitive, accent sensitive</td>
</tr>
<tr>
<td>Character Set</td>
<td>ID</td>
<td>Bytes per Char</td>
<td>Collation</td>
<td>Language</td>
</tr>
<tr>
<td>--------------</td>
<td>----</td>
<td>----------------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>ISO_PLK</td>
<td>Polish</td>
</tr>
<tr>
<td>ISO8859_3</td>
<td>23</td>
<td>1</td>
<td>ISO8859_3</td>
<td>Latin 3 — Southern Europe (Malta, Esperanto)</td>
</tr>
<tr>
<td>ISO8859_4</td>
<td>34</td>
<td>1</td>
<td>ISO8859_4</td>
<td>Latin 4 — Northern Europe (Estonian, Latvian, Lithuanian, Greenlandic, Lappish)</td>
</tr>
<tr>
<td>ISO8859_5</td>
<td>35</td>
<td>1</td>
<td>ISO8859_5</td>
<td>Cyrillic (Russian)</td>
</tr>
<tr>
<td>ISO8859_6</td>
<td>36</td>
<td>1</td>
<td>ISO8859_6</td>
<td>Arabic</td>
</tr>
<tr>
<td>ISO8859_7</td>
<td>37</td>
<td>1</td>
<td>ISO8859_7</td>
<td>Greek</td>
</tr>
<tr>
<td>ISO8859_8</td>
<td>38</td>
<td>1</td>
<td>ISO8859_8</td>
<td>Hebrew</td>
</tr>
<tr>
<td>ISO8859_9</td>
<td>39</td>
<td>1</td>
<td>ISO8859_9</td>
<td>Latin 5</td>
</tr>
<tr>
<td>ISO8859_13</td>
<td>40</td>
<td>1</td>
<td>ISO8859_13</td>
<td>Latin 7 — Baltic</td>
</tr>
<tr>
<td>KOI8R</td>
<td>63</td>
<td>1</td>
<td>KOI8R</td>
<td>Russian — dictionary ordering</td>
</tr>
<tr>
<td>KOI8U</td>
<td>64</td>
<td>1</td>
<td>KOI8U</td>
<td>Ukrainian — dictionary ordering</td>
</tr>
<tr>
<td>KSC_5601</td>
<td>44</td>
<td>2</td>
<td>KSC_5601</td>
<td>Korean</td>
</tr>
<tr>
<td>NEXT</td>
<td>19</td>
<td>1</td>
<td>NEXT</td>
<td>Coding NeXTSTEP</td>
</tr>
<tr>
<td>OCTETS</td>
<td>1</td>
<td>1</td>
<td>OCTETS</td>
<td>Binary character encoding</td>
</tr>
<tr>
<td>SJIS_0208</td>
<td>5</td>
<td>2</td>
<td>SJIS_0208</td>
<td>Japanese</td>
</tr>
<tr>
<td>TIS620</td>
<td>66</td>
<td>1</td>
<td>TIS620</td>
<td>Thai</td>
</tr>
<tr>
<td>TIS620_UNICODE</td>
<td></td>
<td></td>
<td>TIS620_UNICODE</td>
<td>Thai</td>
</tr>
<tr>
<td>Character Set</td>
<td>ID</td>
<td>Bytes per Char</td>
<td>Collation</td>
<td>Language</td>
</tr>
<tr>
<td>----------------</td>
<td>----</td>
<td>----------------</td>
<td>-------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>UNICODE_FSS</td>
<td>3</td>
<td>3</td>
<td>UNICODE_FSS</td>
<td>All English</td>
</tr>
<tr>
<td>UTF8</td>
<td>4</td>
<td>4</td>
<td>UTF8</td>
<td>Any language that is supported in Unicode 4.0</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>UCS_BASIC</td>
<td>Any language that is supported in Unicode 4.0</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>UNICODE</td>
<td>Any language that is supported in Unicode 4.0</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>UNICODE_CI</td>
<td>Any language that is supported in Unicode 4.0 — Case insensitive</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>UNICODE_CI_AI</td>
<td>Any language that is supported in Unicode 4.0 — Case insensitive and accent insensitive</td>
</tr>
<tr>
<td>WIN1250</td>
<td>51</td>
<td>1</td>
<td>WIN1250</td>
<td>ANSI — Central Europe</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>BS_BA</td>
<td>Bosnian</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PXW_CSY</td>
<td>Czech</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PXW_HUN</td>
<td>Hungarian — case insensitive, accent sensitive</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PXW_HUNDC</td>
<td>Hungarian — dictionary ordering</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PXW_PLK</td>
<td>Polish</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PXW_SLOV</td>
<td>Slovenian</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>WIN_CZ</td>
<td>Czech</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>WIN_CZ_CI_AI</td>
<td>Czech — Case insensitive and accent insensitive</td>
</tr>
<tr>
<td>WIN1251</td>
<td>52</td>
<td>1</td>
<td>WIN1251</td>
<td>ANSI Cyrillic</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PXW_CYRL</td>
<td>Paradox Cyrillic (Russian)</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>WIN1251_UA</td>
<td>Ukrainian</td>
</tr>
<tr>
<td>WIN1252</td>
<td>53</td>
<td>1</td>
<td>WIN1252</td>
<td>ANSI — Latin I</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PXW_INTL</td>
<td>English International</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PXW_INTL850</td>
<td>Paradox multilingual Latin I</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PXW_NORDAN4</td>
<td>Norwegian and Danish</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PXW_SPAN</td>
<td>Paradox Spanish</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PXW_SWEDFIN</td>
<td>Swedish and Finnish</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>WIN_PTBR</td>
<td>Portuguese — Brazil</td>
</tr>
<tr>
<td>WIN1253</td>
<td>54</td>
<td>1</td>
<td>WIN1253</td>
<td>ANSI Greek</td>
</tr>
<tr>
<td>Character Set</td>
<td>ID</td>
<td>Bytes per Char</td>
<td>Collation</td>
<td>Language</td>
</tr>
<tr>
<td>---------------</td>
<td>-----</td>
<td>----------------</td>
<td>-------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>WIN1254</td>
<td>55</td>
<td>1</td>
<td>WIN1254</td>
<td>ANSI Turkish</td>
</tr>
<tr>
<td>WIN1255</td>
<td>58</td>
<td>1</td>
<td>WIN1255</td>
<td>ANSI Hebrew</td>
</tr>
<tr>
<td>WIN1256</td>
<td>59</td>
<td>1</td>
<td>WIN1256</td>
<td>ANSI Arabic</td>
</tr>
<tr>
<td>WIN1257</td>
<td>60</td>
<td>1</td>
<td>WIN1257</td>
<td>ANSI Baltic</td>
</tr>
<tr>
<td>WIN1257_EE</td>
<td></td>
<td></td>
<td>WIN1257_EE</td>
<td>Estonian — Dictionary ordering</td>
</tr>
<tr>
<td>WIN1257_LT</td>
<td></td>
<td></td>
<td>WIN1257_LT</td>
<td>Lithuanian — Dictionary ordering</td>
</tr>
<tr>
<td>WIN1257_LV</td>
<td></td>
<td></td>
<td>WIN1257_LV</td>
<td>Latvian — Dictionary ordering</td>
</tr>
<tr>
<td>WIN1258</td>
<td>65</td>
<td>1</td>
<td>WIN1258</td>
<td>Vietnamese</td>
</tr>
</tbody>
</table>
Appendix H: License notice

The contents of this Documentation are subject to the Public Documentation License Version 1.0 (the “License”); you may only use this Documentation if you comply with the terms of this License. Copies of the License are available at https://www.firebirdsql.org/pdfmanual/pdl.pdf (PDF) and https://www.firebirdsql.org/manual/pdl.html (HTML).

The Original Documentation is titled Firebird 3.0 Language Reference. This Documentation was derived from Firebird 2.5 Language Reference.

The Initial Writers of the Original Documentation are: Paul Vinkenoog, Dmitry Yemanov, Thomas Woinke and Mark Rotteveel. Writers of text originally in Russian are Denis Simonov, Dmitry Filippov, Alexander Karpeykin, Alexey Kovyazin and Dmitry Kuzmenko.

Copyright © 2008-2024. All Rights Reserved. Initial Writers contact: paul at vinkenoog dot nl.

Writers and Editors of included PDL-licensed material are: J. Beesley, Helen Borrie, Arno Brinkman, Frank Ingermann, Vlad Khorsun, Alex Peshkov, Nickolay Samofatov, Adriano dos Santos Fernandes, Dmitry Yemanov.

Included portions are Copyright © 2001-2024 by their respective authors. All Rights Reserved.

Contributor(s): Mark Rotteveel.

Portions created by Mark Rotteveel are Copyright © 2018-2024. All Rights Reserved. (Contributor contact(s): mrotteveel at users dot sourceforge dot net).
## Appendix I: Document History

The exact file history is recorded in our *git* repository; see [https://github.com/FirebirdSQL/firebird-documentation](https://github.com/FirebirdSQL/firebird-documentation)

### Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>17 Jan 2024</td>
<td>M</td>
<td>Corrected result type description of <code>SUM</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td>Removed <code>ALTER PACKAGE BODY</code> documentation, as this feature doesn't exist (#200)</td>
</tr>
<tr>
<td>1.1</td>
<td>15 Dec 2023</td>
<td>M</td>
<td>Added types 4 and 5 to <code>RDB$FUNCTION_ARGUMENTS.RDB$MECHANISM</code> (#192)</td>
</tr>
<tr>
<td>1.1</td>
<td>09 Nov 2023</td>
<td>M</td>
<td>Fixed incorrect &quot;equivalent&quot; for <code>REGR_COUNT</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td>Added negative subtype to <code>RDB$FIELDS.RDB$FIELD_SUB_TYPE</code>, and fixed formatting</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Wrong link from <code>ALTER TRIGGER</code> to <code>CREATE TRIGGER</code> (#189)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Added TCPv6 to <code>NETWORK_PROTOCOL</code> for <code>SYSTEM</code> namespace (#170)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fixed description of <code>RDB$INDICES.RDB$FOREIGN_KEY</code> (#191)</td>
</tr>
<tr>
<td>1.1</td>
<td>20 Jun 2023</td>
<td>M</td>
<td>Computed columns can be indexed with expression indexes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td>Replaced firebird-docs references with firebird-devel</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Add caution about relying on ordered derived tables for <code>LIST()</code></td>
</tr>
<tr>
<td>1.1</td>
<td>26 May 2023</td>
<td>M</td>
<td>Added missing context variable names for <code>RDB$GET_CONTEXT()</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td><code>CURRENT_CONNECTION</code> returns <code>BIGINT</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Removed section <code>joins with stored procedures</code> as it no longer applies</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Replaced mention that implicit join is deprecated and might get removed; its use is merely discouraged.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Removed incorrect <code>ROLE</code> keyword from example in <a href="https://github.com/FirebirdSQL/firebird-documentation">Granting the <code>RDB$ADMIN</code> Role in a Regular Database</a></td>
</tr>
<tr>
<td>1.1</td>
<td>10 May 2023</td>
<td>M</td>
<td>Removed incorrect <code>&lt;common-table-expression&gt;</code> production in <code>SELECT</code> syntax</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>30 Jan 2023</td>
<td>M</td>
<td>Fixed typo in collation name <code>UCS_BASIC</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>27 Oct 2022</td>
<td>M</td>
<td>Fixed incorrect whitespace in table names</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td>Added missing table name in <code>FROM</code> in <a href="https://github.com/FirebirdSQL/firebird-documentation">Deterministic functions</a> (#177)</td>
</tr>
<tr>
<td>1.1</td>
<td>31 Jul 2022</td>
<td>M</td>
<td>Move <code>RDB$TRIGGER_TYPE</code> description to separate section to prevent truncation of table cell in PDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>18 Jul 2022</td>
<td>M</td>
<td>Fix documentation for <code>RDB$INDICES.RDB$INDEX_TYPE</code> (#174)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Revision</td>
<td>Date</td>
<td>Change Details</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>1.9</td>
<td>13 Jul 2022</td>
<td>DATEDIFF unit MILLISECOND returns NUMERIC(18,1) since Firebird 3.0.8 (#173)</td>
<td></td>
</tr>
<tr>
<td>1.8</td>
<td>03 Jan 2022</td>
<td>Values for RDB$RELATION_FIELDS.RDB$IDENTITYTYPE were swapped (#168)</td>
<td></td>
</tr>
<tr>
<td>1.7</td>
<td>16 Oct 2021</td>
<td>EXECUTE STATEMENT named parameters are regular identifiers (#164)</td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td>29 Sep 2021</td>
<td>Explicitly document transaction isolation level of ON CONNECT/ON DISCONNECT triggers (#163)</td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>31 Jul 2021</td>
<td>Fix behaviour documented for SNAPSHOT TABLE STABILITY (#158)</td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>23 Jul 2021</td>
<td>Remove extra SELECT in select syntax</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>13 Jun 2021</td>
<td>• Fixed wrong table title NUMERIC → DECIMAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Fixed wrong link title DATEADD → DATEDIFF</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>27 Apr 2021</td>
<td>• Added missing } in regular expression special characters (see issue 124)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Fixed rendering issue hiding the_ in regular expression special characters</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Improve wording of CURRENT_CONNECTION and CURRENT_TRANSACTION (see issue 96)</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>05 Apr 2021</td>
<td>Corrected syntax mistake in SUBSTRING(... SIMILAR ...) documentation.</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>20 Feb 2021</td>
<td>Using the Firebird 2.5 Language Reference as a starting point, incorporated all changes in Firebird 3.0, using the Firebird 3 Release Notes and the Russian Firebird 3.0 Language Reference.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Some restructuring was done for maintainability and readability.</td>
<td></td>
</tr>
</tbody>
</table>