Preface

This volume represents a compilation of topics concerning Firebird’s SQL language written by members of the Russian-speaking community of Firebird developers and users. In 2014, it culminated in a language reference manual, in Russian. At the instigation of Alexey Kovyazin, a campaign was launched amongst Firebird users world-wide to raise funds to pay for a professional translation into English, from which translations into other languages would proceed under the auspices of the Firebird Documentation Project.
# Table of Contents

1. About the Firebird SQL Language Reference: for Firebird 2.5 .......................... 13
   1.1. Subject Matter .......................................................... 13
   1.2. Authorship ............................................................. 13
      1.2.1. Language Reference Updates .................................. 13
      1.2.2. Gestation of the Big Book ................................... 13
      1.2.3. Contributors .................................................... 14
   1.3. Acknowledgments ..................................................... 15
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 .............................................................. 19
      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 ................................................ 29
      3.3.1. NUMERIC .......................................................... 29
      3.3.2. DECIMAL .......................................................... 30
   3.4. Data Types for Dates and Times ....................................... 31
      3.4.1. DATE ............................................................... 32
      3.4.2. TIME ............................................................... 32
      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 ............................................................. 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.3. DROP DOMAIN</td>
<td>103</td>
</tr>
<tr>
<td>5.4. TABLE</td>
<td>103</td>
</tr>
<tr>
<td>5.4.1. CREATE TABLE</td>
<td>104</td>
</tr>
<tr>
<td>5.4.2. ALTER TABLE</td>
<td>119</td>
</tr>
<tr>
<td>5.4.3. DROP TABLE</td>
<td>126</td>
</tr>
<tr>
<td>5.4.4. RECREATE TABLE</td>
<td>127</td>
</tr>
<tr>
<td>5.5. INDEX</td>
<td>127</td>
</tr>
<tr>
<td>5.5.1. CREATE INDEX</td>
<td>128</td>
</tr>
<tr>
<td>5.5.2. ALTER INDEX</td>
<td>131</td>
</tr>
<tr>
<td>5.5.3. DROP INDEX</td>
<td>132</td>
</tr>
<tr>
<td>5.5.4. SET STATISTICS</td>
<td>133</td>
</tr>
<tr>
<td>5.6. VIEW</td>
<td>134</td>
</tr>
<tr>
<td>5.6.1. CREATE VIEW</td>
<td>134</td>
</tr>
<tr>
<td>5.6.2. ALTER VIEW</td>
<td>138</td>
</tr>
<tr>
<td>5.6.3. CREATE OR ALTER VIEW</td>
<td>139</td>
</tr>
<tr>
<td>5.6.4. DROP VIEW</td>
<td>140</td>
</tr>
<tr>
<td>5.6.5. RECREATE VIEW</td>
<td>141</td>
</tr>
<tr>
<td>5.7. TRIGGER</td>
<td>142</td>
</tr>
<tr>
<td>5.7.1. CREATE TRIGGER</td>
<td>142</td>
</tr>
<tr>
<td>5.7.2. ALTER TRIGGER</td>
<td>148</td>
</tr>
<tr>
<td>5.7.3. CREATE OR ALTER TRIGGER</td>
<td>151</td>
</tr>
<tr>
<td>5.7.4. DROP TRIGGER</td>
<td>152</td>
</tr>
<tr>
<td>5.7.5. RECREATE TRIGGER</td>
<td>152</td>
</tr>
<tr>
<td>5.8. PROCEDURE</td>
<td>153</td>
</tr>
<tr>
<td>5.8.1. CREATE PROCEDURE</td>
<td>154</td>
</tr>
<tr>
<td>5.8.2. ALTER PROCEDURE</td>
<td>159</td>
</tr>
<tr>
<td>5.8.3. CREATE OR ALTER PROCEDURE</td>
<td>162</td>
</tr>
<tr>
<td>5.8.4. DROP PROCEDURE</td>
<td>163</td>
</tr>
<tr>
<td>5.8.5. RECREATE PROCEDURE</td>
<td>164</td>
</tr>
<tr>
<td>5.9. EXTERNAL FUNCTION</td>
<td>165</td>
</tr>
<tr>
<td>5.9.1. DECLARE EXTERNAL FUNCTION</td>
<td>165</td>
</tr>
<tr>
<td>5.9.2. ALTER EXTERNAL FUNCTION</td>
<td>168</td>
</tr>
<tr>
<td>5.9.3. DROP EXTERNAL FUNCTION</td>
<td>169</td>
</tr>
<tr>
<td>5.10. FILTER</td>
<td>170</td>
</tr>
<tr>
<td>5.10.1. DECLARE FILTER</td>
<td>170</td>
</tr>
<tr>
<td>5.10.2. DROP FILTER</td>
<td>173</td>
</tr>
<tr>
<td>5.11. SEQUENCE (GENERATOR)</td>
<td>173</td>
</tr>
<tr>
<td>5.11.1. CREATE SEQUENCE (GENERATOR)</td>
<td>174</td>
</tr>
<tr>
<td>5.11.2. ALTER SEQUENCE</td>
<td>175</td>
</tr>
<tr>
<td>5.11.3. SET GENERATOR</td>
<td>175</td>
</tr>
<tr>
<td>5.11.4. DROP SEQUENCE (GENERATOR)</td>
<td>176</td>
</tr>
</tbody>
</table>
### 5.12. EXCEPTION
- 5.12.1. CREATE EXCEPTION .............................................................. 177
- 5.12.2. ALTER EXCEPTION ............................................................... 178
- 5.12.3. CREATE OR ALTER EXCEPTION ............................................... 179
- 5.12.4. DROP EXCEPTION ................................................................. 180
- 5.12.5. RECREATE EXCEPTION .......................................................... 181

### 5.13. COLLATION
- 5.13.1. CREATE COLLATION ............................................................ 181
- 5.13.2. DROP COLLATION ............................................................... 184

### 5.14. CHARACTER SET
- 5.14.1. ALTER CHARACTER SET ...................................................... 185

### 5.15. ROLE
- 5.15.1. CREATE ROLE ................................................................. 186
- 5.15.2. ALTER ROLE ................................................................. 187
- 5.15.3. DROP ROLE ................................................................. 187

### 5.16. COMMENTS
- 5.16.1. COMMENT ON ................................................................. 188

### 6. Data Manipulation (DML) Statements
- 6.1. SELECT ................................................................. 191
  - 6.1.1. FIRST, SKIP .............................................................. 192
  - 6.1.2. The SELECT Columns List .............................................. 194
  - 6.1.3. The FROM clause ....................................................... 198
  - 6.1.4. Joins ................................................................. 205
  - 6.1.5. The WHERE clause ................................................... 214
  - 6.1.6. The GROUP BY clause ................................................. 217
  - 6.1.7. The PLAN clause ..................................................... 222
  - 6.1.8. UNION ................................................................. 227
  - 6.1.9. ORDER BY .............................................................. 230
  - 6.1.10. ROWS ................................................................. 233
  - 6.1.11. FOR UPDATE [OF] .................................................... 236
  - 6.1.12. WITH LOCK .......................................................... 236
  - 6.1.13. INTO ................................................................. 239
  - 6.1.14. Common Table Expressions (“WITH … AS … SELECT”) ....... 241

- 6.2. INSERT ................................................................. 246
  - 6.2.1. INSERT … VALUES ..................................................... 247
  - 6.2.2. INSERT … SELECT ..................................................... 247
  - 6.2.3. INSERT … DEFAULT VALUES ........................................... 249
  - 6.2.4. The RETURNING clause ............................................... 249
  - 6.2.5. Inserting into BLOB columns ......................................... 250

- 6.3. UPDATE ................................................................. 250
  - 6.3.1. Using an alias ....................................................... 252
6.3.2. The SET Clause ........................................... 252
6.3.3. The WHERE Clause ..................................... 253
6.3.4. The ORDER BY and ROWS Clauses .................... 255
6.3.5. The RETURNING Clause ................................. 255
6.3.6. Updating BLOB columns ................................. 256
6.4. UPDATE OR INSERT ........................................ 256
   6.4.1. The RETURNING clause ............................... 258
   6.4.2. Example of UPDATE OR INSERT .................... 258
6.5. DELETE .................................................... 258
   6.5.1. Aliases ............................................... 259
   6.5.2. WHERE ................................................ 260
   6.5.3. PLAN ................................................ 260
   6.5.4. ORDER BY and ROWS ................................. 261
   6.5.5. RETURNING ........................................ 262
6.6. MERGE .................................................... 263
6.7. EXECUTE PROCEDURE ...................................... 265
   6.7.1. “Executable” Stored Procedure ..................... 266
   6.7.2. Examples of EXECUTE PROCEDURE ................. 266
6.8. EXECUTE BLOCK ............................................ 267
   6.8.1. Input and output parameters ....................... 269
   6.8.2. Statement Terminators ............................... 270
7. Procedural SQL (PSQL) Statements .................................. 271
   7.1. Elements of PSQL ....................................... 271
      7.1.1. DML Statements with Parameters ................. 271
      7.1.2. Transactions ..................................... 271
      7.1.3. Module Structure ................................ 271
   7.2. Stored Procedures ...................................... 274
      7.2.1. Benefits of Stored Procedures .................... 274
      7.2.2. Types of Stored Procedures ........................ 274
      7.2.3. Creating a Stored Procedure ..................... 275
      7.2.4. Modifying a Stored Procedure .................... 276
      7.2.5. Deleting a Stored Procedure ..................... 276
   7.3. Stored Functions ....................................... 276
7.4. PSQL Blocks .............................................. 277
7.5. Triggers .................................................. 278
   7.5.1. Firing Order (Order of Execution) .................. 278
   7.5.2. DML Triggers ........................................ 278
   7.5.3. Database Triggers ................................... 279
   7.5.4. Creating Triggers .................................... 280
   7.5.5. Modifying Triggers .................................. 281
   7.5.6. Deleting a Trigger .................................. 281
7.6. Writing the Body Code ................................................. 282
  7.6.1. Assignment Statements ........................................ 282
  7.6.2. DECLARE CURSOR ............................................. 283
  7.6.3. DECLARE VARIABLE ........................................... 286
  7.6.4. BEGIN … END .................................................. 289
  7.6.5. IF … THEN … ELSE ............................................ 292
  7.6.6. WHILE … DO ................................................... 293
  7.6.7. LEAVE .......................................................... 295
  7.6.8. EXIT ............................................................ 297
  7.6.9. SUSPEND ......................................................... 297
  7.6.10. EXECUTE STATEMENT ........................................ 298
  7.6.11. FOR SELECT .................................................. 305
  7.6.12. FOR EXECUTE STATEMENT .................................... 308
  7.6.13. OPEN .......................................................... 309
  7.6.14. FETCH ........................................................ 312
  7.6.15. CLOSE ........................................................ 313
  7.6.16. IN AUTONOMOUS TRANSACTION ............................. 314
  7.6.17. POST EVENT ................................................... 315
7.7. Trapping and Handling Errors ...................................... 316
  7.7.1. System Exceptions ............................................. 317
  7.7.2. Custom Exceptions ............................................ 317
  7.7.3. EXCEPTION .................................................... 317
  7.7.4. WHEN … DO ................................................... 320
8. Built-in Functions ....................................................... 324
  8.1. Context Functions ................................................ 324
    8.1.1. RDB$GET_CONTEXT() ......................................... 324
    8.1.2. RDB$SET_CONTEXT() ......................................... 326
  8.2. Mathematical Functions ......................................... 327
    8.2.1. ABS() ......................................................... 327
    8.2.2. ACOS() ....................................................... 328
    8.2.3. ASIN() ....................................................... 328
    8.2.4. ATAN() ....................................................... 329
    8.2.5. ATAN2() ..................................................... 329
    8.2.6. CEIL(), CEILING() .......................................... 330
    8.2.7. COS() ......................................................... 331
    8.2.8. COSH() ....................................................... 331
    8.2.9. COT() ......................................................... 332
    8.2.10. EXP() ......................................................... 332
    8.2.11. FLOOR() ..................................................... 333
    8.2.12. LN() ........................................................ 333
    8.2.13. LOG() ......................................................... 334
<table>
<thead>
<tr>
<th>Table of Contents</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$EXCEPTIONS</td>
<td>494</td>
</tr>
<tr>
<td>RDB$FIELDS</td>
<td>494</td>
</tr>
<tr>
<td>RDB$FIELD_DIMENSIONS</td>
<td>498</td>
</tr>
<tr>
<td>RDB$FILES</td>
<td>498</td>
</tr>
<tr>
<td>RDB$FILTERS</td>
<td>499</td>
</tr>
<tr>
<td>RDB$FORMATS</td>
<td>499</td>
</tr>
<tr>
<td>RDB$FUNCTIONS</td>
<td>500</td>
</tr>
<tr>
<td>RDB$FUNCTION_ARGUMENTS</td>
<td>501</td>
</tr>
<tr>
<td>RDB$GENERATORS</td>
<td>502</td>
</tr>
<tr>
<td>RDB$INDICES</td>
<td>502</td>
</tr>
<tr>
<td>RDB$INDEX_SEGMENTS</td>
<td>504</td>
</tr>
<tr>
<td>RDB$LOG_FILES</td>
<td>504</td>
</tr>
<tr>
<td>RDB$PAGES</td>
<td>504</td>
</tr>
<tr>
<td>RDB$PROCEDURES</td>
<td>505</td>
</tr>
<tr>
<td>RDB$PROCEDURE_PARAMETERS</td>
<td>506</td>
</tr>
<tr>
<td>RDB$REF_CONSTRAINTS</td>
<td>507</td>
</tr>
<tr>
<td>RDB$RELATIONS</td>
<td>508</td>
</tr>
<tr>
<td>RDB$RELATION_CONSTRAINTS</td>
<td>509</td>
</tr>
<tr>
<td>RDB$RELATION_FIELDS</td>
<td>510</td>
</tr>
<tr>
<td>RDB$ROLES</td>
<td>511</td>
</tr>
<tr>
<td>RDB$SECURITY_CLASSES</td>
<td>511</td>
</tr>
<tr>
<td>RDB$TRANSACTIONS</td>
<td>512</td>
</tr>
<tr>
<td>RDB$TRIGGERS</td>
<td>512</td>
</tr>
<tr>
<td>RDB$TRIGGER_MESSAGES</td>
<td>514</td>
</tr>
<tr>
<td>RDB$TYPES</td>
<td>514</td>
</tr>
<tr>
<td>RDB$USER_PRIVILEGES</td>
<td>515</td>
</tr>
<tr>
<td>RDB$VIEW_RELATIONS</td>
<td>515</td>
</tr>
<tr>
<td>Appendix E: Monitoring Tables</td>
<td>517</td>
</tr>
<tr>
<td>MON$ATTACHMENTS</td>
<td>518</td>
</tr>
<tr>
<td>Using MON$ATTACHMENTS to Kill a Connection</td>
<td>519</td>
</tr>
<tr>
<td>MON$CALL_STACK</td>
<td>519</td>
</tr>
<tr>
<td>MON$CONTEXT_VARIABLES</td>
<td>521</td>
</tr>
<tr>
<td>MON$DATABASE</td>
<td>521</td>
</tr>
<tr>
<td>MON$IO_STATS</td>
<td>522</td>
</tr>
<tr>
<td>MON$MEMORY_USAGE</td>
<td>523</td>
</tr>
<tr>
<td>MON$RECORD_STATS</td>
<td>524</td>
</tr>
<tr>
<td>MON$STATEMENTS</td>
<td>524</td>
</tr>
<tr>
<td>Using MON$STATEMENTS to Cancel a Query</td>
<td>525</td>
</tr>
<tr>
<td>MON$TRANSACTIONS</td>
<td>525</td>
</tr>
<tr>
<td>Appendix F: Character Sets and Collation Sequences</td>
<td>527</td>
</tr>
<tr>
<td>Appendix G: License notice</td>
<td>533</td>
</tr>
</tbody>
</table>
Chapter 1. About the Firebird SQL Language Reference: for Firebird 2.5

This Firebird SQL Language Reference is the first 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. It has a long history.

1.1. Subject Matter

The subject matter 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.

1.2. Authorship

The material for assembling this Language Reference has been accumulating in the tribal lore of the open source community of Firebird core developers and user-developers for 15 years. The gift of the InterBase 6 open source codebase in July 2000 from the (then) Inprise/Borland conglomerate was warmly welcomed. However, it came without rights to existing documentation. Once the code base had been forked by its owners for private, commercial development, it became clear that the open source, non-commercial Firebird community would never be granted right of use.

The two important books from the InterBase 6 published set were the Data Definition Guide and the Language Reference. The former covered the data definition language (DDL) subset of the SQL language, while the latter covered most of the rest. Fortunately for Firebird users over the years, both have been easy to find on-line as PDF books.

1.2.1. Language Reference Updates

The Data Definition Guide, covering the creation and maintenance of metadata for databases, was “good enough” for several years: the data definition language (DDL) of a DBMS is stable and grows slowly in comparison to the data manipulation language (DML) employed for queries and the PSQL used for server-based programming.

The leader of the Firebird Project’s documentation team, Paul Vinkenoog, took up the cause for documenting the huge volume of improvements and additions to DML and PSQL as Firebird advanced through its releases. Paul was personally responsible for collating, assembling and, to a great extent, authoring a cumulative series of “Language Reference Updates” — one for every major release from v.1.5 forward.

1.2.2. Gestation of the Big Book

From around 2010, Paul, with Firebird Project lead Dmitry Yemanov and a documenter colleague
Thomas Woinke, set about the task of designing and assembling a complete SQL language reference for Firebird. They began with the material from the LangRef Updates, which is voluminous. It was going to be a big job but, for all concerned, a spare-time one.

Then, in 2013-4, two benefactor companies — MICEX and IBSurgeon — funded three writers to take up this stalled book outline and publish a Firebird 2.5 Language Reference in Russian. They wrote the bulk of the missing DDL section from scratch and wrote, translated or reused DML and PSQL material from the LangRef Updates, Russian language support forums, Firebird release notes, readme files and other sources. By the end of 2014, they had the task almost complete, in the form of a Microsoft Word document.

Translation …

The Russian sponsors, recognising that their efforts needed to be shared with the world-wide Firebird community, asked some Project members to initiate a crowd-funding campaign to have the Russian text professionally translated into English. The translated text would be edited and converted to the Project’s standard DocBook format for addition to the open document library of the Firebird Project. From there, the source text would be available for translation into other languages for addition to the library.

The fund-raising campaign happened at the end of 2014 and was successful. In June, 2015, professional translator Dmitry Borodin began translating the Russian text. From him, the raw English text went in stages for editing and DocBook conversion by Helen Borrie — and here is The Firebird SQL Language Reference for V.2.5, by…everyone!

… and More Translation

Once the DocBook source appears in CVS, we hope the trusty translators will start making versions in German, Japanese, Italian, French, Portuguese, Spanish, Czech. Certainly, we never have enough translators so please, you Firebirders who have English as a second language, do consider translating some sections into your first language.

1.2.3. Contributors

Direct Content

• Dmitry Filippov (writer)
• Alexander Karpeykin (writer)
• Alexey Kovyazin (writer, editor)
• Dmitry Kuzmenko (writer, editor)
• Denis Simonov (writer, editor, coordinator)
• Paul Vinkenoog (writer, designer)
• Dmitry Yemanov (writer)

Resource Content

• Adriano dos Santos Fernandes
1.3. Acknowledgments

The first full language reference manual for Firebird would not have eventuated without the funding that finally brought it to fruition. We acknowledge these contributions with gratitude and thank you all for stepping up.

Sponsors and Other 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.*

Sponsors of the Translation Project

Syntess Software BV (Netherlands)

Mitaro Business Solutions (Liechtenstein)

Other Donors

Listed below are the names of companies and individuals whose cash contributions covered the costs for translation into English, editing of the raw, translated text and conversion of the whole into the Firebird Project’s standard DocBook 4 documentation source format.
<table>
<thead>
<tr>
<th>Company/Individual</th>
<th>Name/Title</th>
<th>Company/Individual</th>
<th>Name/Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrity Software Design, Inc.</td>
<td>dimari GmbH (Germany)</td>
<td>beta Eigenheim GmbH (Germany)</td>
<td></td>
</tr>
<tr>
<td>KIMData GmbH (Germany)</td>
<td>Jason Wharton (U.S.A)</td>
<td>Trans-X (Sweden)</td>
<td></td>
</tr>
<tr>
<td>Sanchez Balcewich (Uruguay)</td>
<td>Cointec Ingenieros y Consultores, S.L. (Spain)</td>
<td>Aage Johansen (Norway)</td>
<td></td>
</tr>
<tr>
<td>Mattic Software (Netherlands)</td>
<td>Andre Knappstein (Germany)</td>
<td>Paul F. McGuire (U.S.A.)</td>
<td></td>
</tr>
<tr>
<td>Marcus Marques da Rocha (Brazil)</td>
<td>Martin Kerkhoff</td>
<td>Thomas Vedel (Denmark)</td>
<td></td>
</tr>
<tr>
<td>Bulhan Bulhan (Australia)</td>
<td>Alexandre Benson Smith (Brazil)</td>
<td>Guillermo Nabrink (Brazil)</td>
<td></td>
</tr>
<tr>
<td>Pierre Voirin (France)</td>
<td>Heiko Tappe (Germany)</td>
<td>Doug Chamberlin (U.S.A.)</td>
<td></td>
</tr>
<tr>
<td>Craig Cox (U.S.A.)</td>
<td>OMNet, Inc. (Switzerland)</td>
<td>Alfred Steller (Germany)</td>
<td></td>
</tr>
<tr>
<td>Konrad Butz (Germany)</td>
<td>Thomas Smekal (Canada)</td>
<td>Carlos H. Cantu (Brazil)</td>
<td></td>
</tr>
<tr>
<td>XTRALOG SARL (France)</td>
<td>Laszlo Urmenyi (Brazil)</td>
<td>Fernando Pimenta (Brazil)</td>
<td></td>
</tr>
<tr>
<td>Rudolf Grauberger (Germany)</td>
<td>Thomas Steinmaurer (Austria)</td>
<td>Rene Lobsiger (Switzerland)</td>
<td></td>
</tr>
<tr>
<td>Hian Pin Tjioe</td>
<td>Xavier Codina</td>
<td>Mick Arundell (Australia)</td>
<td></td>
</tr>
<tr>
<td>Russell Belding (New Zealand)</td>
<td>Anticlei Scheid (Brazil)</td>
<td>Luca Minuti (Italy)</td>
<td></td>
</tr>
<tr>
<td>Mark Rotteveel (Netherlands)</td>
<td>Chris Mathews (U.S.A.)</td>
<td>Hannes Streicher (Germany)</td>
<td></td>
</tr>
<tr>
<td>Wolfgang Lemermeyer (Germany)</td>
<td>Paolo Sciarrini (Italy)</td>
<td>Acosta Belzusarri</td>
<td></td>
</tr>
<tr>
<td>Daniel Motos Guerra</td>
<td>Alberto Alfonso Luna</td>
<td>Simeon Bodurov</td>
<td></td>
</tr>
<tr>
<td>Cees Meijer</td>
<td>Robert Nixon</td>
<td>Olivier Dehorter (France)</td>
<td></td>
</tr>
<tr>
<td>Andras Omacht (Hungary)</td>
<td>Web Express</td>
<td>Sergio Gonzalez</td>
<td></td>
</tr>
<tr>
<td>Marc Bleuwart</td>
<td>Gabor Boros</td>
<td>Shaymon Gracia Campos</td>
<td></td>
</tr>
<tr>
<td>Cserna Zsombor (Hungary)</td>
<td>David Keith</td>
<td>Uwe Gerold</td>
<td></td>
</tr>
<tr>
<td>Daniele Teti (Italy)</td>
<td>Pedro Pablo Busto Criado</td>
<td>Istvan Szabo</td>
<td></td>
</tr>
<tr>
<td>Spiridon Pavlovic</td>
<td>J. L. Garcia Naranjo</td>
<td>A. Morales Morales</td>
<td></td>
</tr>
<tr>
<td>Helen Cullen (New Zealand)</td>
<td>Francisco Ibarra Ozuna</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

- **In 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.

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.

```sql
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.

```sql
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. 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:

\[
\text{fullname} \\
\text{FULLNAME} \\
\text{FuLlNaMe} \\
\text{FullName}
\]

**Regular name syntax**

\[
\begin{align*}
\text{<name>} & ::= \text{<letter>} | \text{<name><letter>} | \text{<name><digit>} | \text{<name>_} | \text{<name>$}
\end{align*}
\]

\[
\begin{align*}
\text{<letter>} & ::= \text{<upper letter>} | \text{<lower letter>}
\end{align*}
\]

\[
\begin{align*}
\text{<upper letter>} & ::= \text{A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z}
\end{align*}
\]

\[
\begin{align*}
\text{<lower letter>} & ::= \text{a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z}
\end{align*}
\]

\[
\begin{align*}
\text{<digit>} & ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\end{align*}
\]

### 2.3.2. Rules for Delimited Object Identifiers

- Length cannot exceed 31 characters
- The entire string must be enclosed in double-quotes, e.g. "anIdentifier"
- It may contain characters from any Latin 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
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 all regular names 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 represent data in a direct format. Examples of standard types of literals are:

- integer       - 0, -34, 45, 0X080000000;
- real          - 0.0, -3.14, 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';
- 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.

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.
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

```
<comment> ::= <block comment> | <single-line comment>

<block comment> ::= /* <ASCII char>[<ASCII char> …] */

<single-line comment> ::= -- <ASCII char>[<ASCII char> …]<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

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 a dynamically 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>CHAR((n)), CHARACTER((n))</td>
<td>(n) characters. Size in bytes depends on the encoding, the number of bytes in a character</td>
<td>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>
</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><strong>DECIMAL (precision, scale)</strong></td>
<td>Varying (16, 32 or 64 bits)</td>
<td>precision = from 1 to 18, defines the least possible number of digits to store; scale = from 0 to 18, defines the number of digits after the decimal point</td>
<td>A number with a decimal point that has scale digits after the point. scale must be less than or equal to precision. Example: DECIMAL(10,3) contains a number in exactly the following format: pppppp.sss</td>
</tr>
<tr>
<td><strong>DOUBLE PRECISION</strong></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><strong>FLOAT</strong></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><strong>INTEGER, INT</strong></td>
<td>32 bits</td>
<td>-2,147,483,648 up to 2,147,483,647</td>
<td>Signed long</td>
</tr>
<tr>
<td><strong>NUMERIC (precision, scale)</strong></td>
<td>Varying (16, 32 or 64 bits)</td>
<td>precision = from 1 to 18, defines the exact number of digits to store; scale = from 0 to 18, defines the number of digits after the decimal point</td>
<td>A number with a decimal point that has scale digits after the point. scale must be less than or equal to precision. Example: NUMERIC(10,3) contains a number in exactly the following format: pppppp.sss</td>
</tr>
<tr>
<td><strong>SMALLINT</strong></td>
<td>16 bits</td>
<td>-32,768 to 32,767</td>
<td>Signed short (word)</td>
</tr>
<tr>
<td><strong>TIME</strong></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><strong>TIMESTAMP</strong></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>
</tbody>
</table>
### Name, Size, Precision & Limits

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Precision &amp; Limits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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.
**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*. 

```sql
CREATE TABLE CUSTOMER (  
    CUST_NO INTEGER NOT NULL,  
    CUSTOMER VARCHAR(25) NOT NULL,  
    CONTACT_FIRST VARCHAR(15),  
    CONTACT_LAST VARCHAR(20),  
    ...  
    PRIMARY KEY (CUST_NO) )
```
CREATE TABLE WHOLELOTTARECORDS (  
  ID BIGINT NOT NULL PRIMARY KEY,  
  DESCRIPTION VARCHAR(32)  
);

INSERT INTO MYBIGINTS VALUES (  
  -23645287458723,  
  32832607832,  
  22,  
  -56786237632476,  
  0X6F55A09D42, -- 478177959234  
  0X7FFFFFFFFFFFFFFF, -- 9223372036854775807  
  0XFFFFFFFFFFFFFFFF, -- -1  
  0X8000000, -- -2147483648, an INTEGER  
  0X080000000, -- 2147483648, a BIGINT  
  0XFFFFFFFF, -- -1, an INTEGER  
  0X0FFFFFFFF -- 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 0x8000000 (8 digits) and 0x080000000 (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>
<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 ten-thousandths 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 decimilliseconds 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)
- CURRENT_TIMESTAMP millisecond precision. Precision from seconds to milliseconds can be specified with CURRENT_TIMESTAMP (0|1|2|3)
- Literal 'NOW': millisecond precision
- Functions DATEADD() and DATEDIFF() support up to milliseconds precision. Decimilliseconds can be specified but they are rounded to the nearest integer before any operation is performed
- The EXTRACT() function returns up to decimillisecond precision with the SECOND and MILLISECOND arguments
- For TIME and TIMESTAMP literals, Firebird happily accepts up to decimillisecond 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$database
- the ‘+’ and ‘-’ operators work with decimillisecond precision, but only within the expression. As soon as something is stored or even just SELECTed from RDB$DATABASE, it reverts to millisecond 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()

```sql
EXTRACT (HOUR FROM DATE_FIELD)
EXTRACT (MINUTE FROM DATE_FIELD)
EXTRACT (SECOND 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 DECIMAL(9, 0)</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 DECIMAL(9, 4)</td>
</tr>
<tr>
<td>TIME</td>
<td>-</td>
<td>Numeric value n</td>
<td>TIME reduced by n seconds. The fractional part is taken into account</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>-</td>
<td>TIMESTAMP</td>
<td>Number of days and part-day, within the range DECIMAL(18, 9)</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>-</td>
<td>Numeric value n</td>
<td>TIMESTAMP where the date will decrease by the number of days and part of a day represented by number n — so “- 2.25” will decrease the date by 2 days and 6 hours</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:

```sql
...  
  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:

```sql
...  
  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>
<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>
</tbody>
</table>
Collation | Characteristics
--- | ---
UTF8 | The default, binary collation, identical to UCS_BASIC, which was added for SQL compatibility
UNICODE_CI | Case-insensitive collation, works without taking character case into account. Added in Firebird 2.1
UNICODE_CI_AI | Case-insensitive, accent-insensitive collation, works alphabetically without taking character case or accents into account. Added in Firebird 2.5

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).

```sql
...  
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):

```
max_char_length = FLOOR((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>
<thead>
<tr>
<th>Page Size</th>
<th>Bytes per character</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>4,096</td>
<td>1,015</td>
</tr>
<tr>
<td>8,192</td>
<td>2,039</td>
</tr>
<tr>
<td>16,384</td>
<td>4,087</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) ]
[CHARACTER SET <set>] [COLLATE <name>]
```

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.

**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 VARYING | CHARACTER VARYING } (length)
[CHARACTER SET <set>] [COLLATE <name>]
```
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) ]

A similar data type is available for the variable-length string type: NATIONAL CHARACTER VARYING.

3.6. 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>)

Segment Size

Specifying the BLOB segment 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.

3.6.1. 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.

3.6.2. 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:

\[
\begin{align*}
= & \quad \text{(assignment)} \\
\ne, \ne, \lt, \le, \gt, \ge & \quad \text{(comparison)} \\
\| & \quad \text{(concatenation)} \\
\text{BETWEEN,} & \quad \text{IS [NOT] DISTINCT FROM,} \\
\text{IN,} & \quad \text{ANY | SOME,} \\
\text{ALL} & \\
\end{align*}
\]

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:

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

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

3.6.3. 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
multidimensional 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:
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.

### 3.7. Special Data Types

“Special” data types ...
### 3.7.1. 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:

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

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

```sql
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.

```sql
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.

### 3.8. 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.

#### 3.8.1. Explicit Data Type Conversion

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

**Syntax**

```
CAST ( { <value> | NULL } AS <data_type>)
```

```
<data_type> ::=  
<sql_datatype>  
| [TYPE OF] domain  
| TYPE OF COLUMN relname.colname
```

**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</td>
</tr>
<tr>
<td>BLOB</td>
<td>[VAR]CHAR, BLOB, Numeric types, DATE, TIME, TIMESTAMP</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>
</tbody>
</table>

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:
<timestamp_format> ::= 

<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 
| MM<p>DD<p>YYYY 
| DD<p>MM<p>YYYY 
| NOW 
| TODAY 
| TOMORROW 
| YESTERDAY 
} 

<time_format> := 
| NOW 
| NOW 

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

<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>YYYY</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>
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('04.12.2014' as date) as dt1,
    -- DD.MM.YYYY HH:mm
    cast('04.12.2014 11:37' as timestamp) as dt1,
    -- DD.MM.YYYY HH:mm:ss
    cast('04/12/2014 11:37:12' as timestamp) as dt2,
    -- MM/DD/YYYY HH:mm:ss.nnnn
    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
from rdb$database
```

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.

Syntax

```
<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 = TIMESTAM '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);
```

3.8.2. 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:

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

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:

```
UPDATE ATABLE
SET ADATE = CAST ('25.12.2016' AS DATE) + 1
```

or, with the short cast:
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,

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:

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
```

### 3.9. 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.

#### 3.9.1. 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.

### 3.9.2. 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>
<tr>
<td>CHECK constraints</td>
<td>Yes</td>
<td>To add new conditions to the check, you can use the corresponding CHECK clauses in the CREATE and ALTER statements at the table level.</td>
</tr>
<tr>
<td>NOT NULL</td>
<td>No</td>
<td>Often it is better to leave domain nullable in its definition and decide whether to make it NOT NULL when using the domain to define columns.</td>
</tr>
</tbody>
</table>

### 3.9.3. Creating and Administering Domains

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

---

50
**Short Syntax**

```sql
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
- delete the current default value
- set a new default value
- delete an existing `CHECK` constraint
- add a new `CHECK` constraint

**Short Syntax**

```sql
ALTER DOMAIN name
[{{TO <new_name>}}]
[{{SET DEFAULT {<literal> | NULL | <context_var>} | DROP DEFAULT}}]
[{{ADD [CONSTRAINT] CHECK (<dom_condition>) | DROP CONSTRAINT}}]
[{{TYPE <datatype>}}]
```

When planning to alter 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.
Deleting (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.
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 assertions 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 assertions.

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

Comparison predicates
LIKE, STARTING WITH, CONTAINING, SIMILAR TO, BETWEEN, IS [NOT] NULL 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
A number or a string literal enclosed in apostrophes.
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. Constants

A 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 Constants (Literals)**

A string constant—or string literal—is a series of characters enclosed between a pair of apostrophes (“single quotes”). The maximum length of a string is 32,767 bytes; 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. SQL reserves a different purpose for them.
- 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 VARCHAR column. The maximum length for a VARCHAR 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 Constants 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

```sql
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)

select _utf8 x'53C3A46765' from rdb$database
-- returns 'Säge' (4 chars, 5 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.

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

```sql
INSERT INTO People
VALUES (_ISO8859_1 'Hans-Jörg Schäfer')
```

Number Constants (Literals)

A number constant — or 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 (., full-stop, dot) character and 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.
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 <hexdigit>
\(<\text{hexdigit}>\) ::= one of 0..9, A..F, a..f

Examples

```sql
select 0x6FAA0D3 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$database         -- returns 2655328488 (a BIGINT)
select 0x28ED678A4C987 from rdb$database     -- returns 720001751632263
select 0xFFFFFFFFFFFFFFFF from rdb$database  -- returns -1
```

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).
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 12. 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 13. 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>
</tbody>
</table>
### Comparison Operators

**Table 14. Comparison Operator Precedence**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Is equal to, is identical to</td>
<td>1</td>
</tr>
<tr>
<td>&lt;&gt;, !=, ~=, ^=</td>
<td>Is not equal to</td>
<td>1</td>
</tr>
<tr>
<td>&gt;</td>
<td>Is greater than</td>
<td>1</td>
</tr>
<tr>
<td>&lt;</td>
<td>Is less than</td>
<td>1</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Is greater than or equal to</td>
<td>1</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Is less than or equal to</td>
<td>1</td>
</tr>
<tr>
<td>!&gt;, ~&gt; , ^&gt;</td>
<td>Is not greater than</td>
<td>1</td>
</tr>
<tr>
<td>!&lt;, &lt;=, ^&lt;</td>
<td>Is not less than</td>
<td>1</td>
</tr>
</tbody>
</table>

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

**Example**

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

**See also**

**Other Comparison Predicates.**

### Logical Operators

**Table 15. 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>Operator</td>
<td>Purpose</td>
<td>Precedence</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>------------</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**

DSQL, PSQL

NEXT VALUE FOR returns the next value of a sequence. SEQUENCE is an SQL-compliant term for 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.

**Syntax for NEXT VALUE FOR**

```sql
NEXT VALUE FOR sequence-name
```

**Example**

```sql
NEW.CUST_ID = NEXT VALUE FOR CUSTSEQ;
```

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.

**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.
The CASE construct returns a single value from a number of possible ones. 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**

```sql
CASE <test-expr>
  WHEN <expr> THEN <result>
  [WHEN <expr> THEN <result> ...]
  [ELSE <defaultresult>]
END
```

When this variant is used, test-expr is compared expr 1, expr 2 etc., 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**

```sql
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.
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

```
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.*

**Important to Note**

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:
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

Up to and including Firebird 2.5.x, there is no implementation for a logical (Boolean) data type—that is coming in Firebird 3. However, there are logical expressions (predicates) that can return true, false or unknown.

**Examples**

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

```sql
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.

**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:

```
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 $\neg P$ will return TRUE. On the other hand, if $P$ returns NULL (unknown), then $\neg P$ 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.

#### 4.2.1. Assertions

An assertion is a statement about the data that, like a predicate, can resolve to TRUE, FALSE or NULL. Assertions consist of one or more predicates, possibly negated using $\neg$ and connected by AND and 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 of the ultimate assertion 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:

```
=, >, <, >=, <=, <>
```

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`:

   ```sql
   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:

   ```sql
   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*

DSQL, PSQL, ESQL
Syntax

\[
\text{<value> [NOT] BETWEEN <value_1> AND <value_2>}
\]

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:

\[
\text{<value> >= <value_1> AND <value> <= <value_2>}
\]

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

```
SELECT *
FROM EMPLOYEE
WHERE HIRE_DATE BETWEEN date '01.01.1992' AND CURRENT_DATE
```

LIKE

Available

DSQL, PSQL, ESQL

Syntax

\[
\text{<match value> [NOT] LIKE <pattern>}
\]

\[
\text{[ESCAPE <escape character>]}
\]

\[
\text{<match value> ::= character-type expression}
\]

\[
\text{<pattern> ::= search pattern}
\]

\[
\text{<escape character> ::= escape character}
\]

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.

   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
DSQL, PSQL, ESQL

Syntax

```
<value> [NOT] STARTING WITH <value>
```

The STARTING WITH predicate searches for a string or a string-like type that starts with the characters in its value argument. The search is case-sensitive.

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”:

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

See also
LIKE
CONTAINING

Available
DSQL, PSQL, ESQL
Syntax

```
<value> [NOT] CONTAINING <value>
```

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.

When CONTAINING 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.

**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**

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> ::=<escape-char> <special character> | <escape-char> <escape-char>
<special character> ::= any of the characters []()|^-+*%_.{}
<non-escaped character> ::= any character that is not a <special character>
and not equal to <escape-char> (if defined)
<character class> ::= '.' | '[' <member> ... ']' | '{' <non-member> ... '}'
| '[' <member> ... '^' <non-member> ... ']'
<member>, <non-member> ::= <character> | <range> | <predefined class>
<range> ::= <character> - <character>
<predefined class> ::= '[' <predefined class name> ']' 
<predefined class name> ::= ALPHA | UPPER | LOWER | DIGIT | ALNUM | SPACE | WHITESPACE

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 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:

```
'Birne' similar to 'B_rne'   -- true
'Birne' similar to 'B_ne'    -- false
'Birne' similar to 'B%ne'    -- true
'Birne' similar to 'Bir%ne%' -- true
'Birne' similar to 'Birr%ne' -- false
```

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:

```
'Citroen' similar to 'Cit[arju]oen'     -- true
'Citroen' similar to 'Ci[tr]oen'        -- false
'Citroen' similar to 'Ci[tr][tr]oen'    -- true
```

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.
Predefined Character Classes

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

`:ALPHA:`
Latin letters a..z and A..Z. With an accent-insensitive collation, this class also matches accented forms of these characters.

`:DIGIT:`
Decimal digits 0..9.

`:ALNUM:`
Union of `[:ALPHA:]` and `[:DIGIT:]`.

`:UPPER:`
Uppercase Latin letters A..Z. Also matches lowercase with case-insensitive collation and accented forms with accent-insensitive collation.

`:LOWER:`
Lowercase Latin letters a..z. Also matches uppercase with case-insensitive collation and accented forms with accent-insensitive collation.

`:SPACE:`
Matches the space character (ASCII 32).

`:WHITESPACE:`
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.

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:

'Hallon' similar to 'Hal?on' -- false
'Hallon' similar to 'Hal?lon' -- true
'Hallon' similar to 'Hallo?on' -- true
'Hallon' similar to 'Hallo?l?on' -- false
'Hallon' similar to 'Hallx?lon' -- true
'Hallon' similar to 'H[a-c]?llon[x-z]?' -- true

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

'Icaque' similar to 'Ica*que' -- true
'Icaque' similar to 'Icar*que' -- true
'Icaque' similar to 'I[a-c]*que' -- true
'Icaque' similar to '*e' -- true
'Icaque' similar to '[:ALPHA:]*e' -- true
'Icaque' similar to 'Ica[xyz]*e' -- false

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Mandarijn' similar to 'M[a-p]{2,5}rijn'</td>
<td>true</td>
</tr>
<tr>
<td>'Mandarijn' similar to 'M[a-p]{2,3}rijn'</td>
<td>false</td>
</tr>
<tr>
<td>'Mandarijn' similar to 'M[a-p]{2,3}arijn'</td>
<td>true</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Nektarin' similar to 'Nek</td>
<td>tarin'</td>
</tr>
<tr>
<td>'Nektarin' similar to 'Nektarin</td>
<td>Persika'</td>
</tr>
<tr>
<td>'Nektarin' similar to 'M_+</td>
<td>N_+</td>
</tr>
</tbody>
</table>

**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.
expression in its own right. It can contain all the elements allowed in a regular expression, and can also have quantifiers added to it.

```plaintext
'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:

```plaintext
'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**

DSQL, PSQL

**Syntax**

```plaintext
<operand1> IS [NOT] DISTINCT FROM <operand2>
```

Two operands are considered *DISTINCT* if they have a different value or if one of them is *NULL* and the other non-null. They are *NOT DISTINCT* if they have the same value or if both of them are *NULL*.

**See also**

**IS [NOT] NULL**

**Available**

DSQL, PSQL, ESQL

**Syntax**

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

Since *NULL* is not a value, these operators are not comparison operators. The *IS [NOT] NULL* predicate tests the assertion 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 assertion, returning **TRUE** if the existence or non-existence is confirmed or **FALSE** otherwise.

**EXISTS**

*Available*

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.

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

IN
Available
DSQL, PSQL, ESQL
Syntax

\[
\text{<value> [NOT] IN (<select_stmt> | <value_list>)}
\]
\[
\text{<value_list> ::= <value_1> [, <value_2> …]}
\]

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:

\[
(\text{<value> = <value_1> [OR <value> = <value_2> …]})
\]
\[
\text{<value> = \{ ANY | SOME \} (<select_stmt>)}
\]

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 IN (SELECT model 
                FROM product 
                WHERE maker = 'A');
```

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

```sql
-- 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.

Advice
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”:

   SELECT *
   FROM EMPLOYEE
   WHERE FIRST_NAME IN ('Pete', 'Ann', '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
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 assertion is true. It is
not a numeric quantity, but a logical one that connects the assertion 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:

```
<value expression> <comparison operator> <quantifier> <subquery>
```

**ALL**

Available
DSQL, PSQL, ESQL

Syntax

```
<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.
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*

DSQL, PSQL, ESQL

*Syntax*

```
<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*

```sql
CREATE {DATABASE | SCHEMA} <filespec>
  [USER 'username' [PASSWORD 'password']]  
  [PAGE_SIZE [=] size]  
  [LENGTH [=] num [PAGE[S]]]  
  [SET NAMES 'charset']  
  [DEFAULT CHARACTER SET default_charset]  
  [COLLATION collation]] -- not supported in ESQL  
  [<sec_file> [<sec_file> ...]]  
  [DIFFERENCE FILE 'diff_file'] -- not supported in ESQL
```

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

*<server_spec> ::= servername//[port|service]: | \servername\`

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

<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>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>server_spec</td>
<td>Remote server specification in TCP/IP or Windows Networking style. Optionally includes a port number or service name</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 aliases.conf file</td>
</tr>
<tr>
<td>servername</td>
<td>Host name or IP address of the server where the database is to be created</td>
</tr>
<tr>
<td>username</td>
<td>User name of the owner of the new database. It may consist of up to 31 characters. Case-insensitive</td>
</tr>
<tr>
<td>password</td>
<td>Password of the user name as the database owner. The maximum length is 31 characters; however only the first 8 characters are considered. Case-sensitive</td>
</tr>
<tr>
<td>size</td>
<td>Page size for the database, in bytes. Possible values are 4096 (the default), 8192 and 16384</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>
<tr>
<td>default_charset</td>
<td>Specifies the default character set for string data types</td>
</tr>
<tr>
<td>collation</td>
<td>Default collation for the default character set</td>
</tr>
<tr>
<td>sec_file</td>
<td>File specification for a secondary file</td>
</tr>
<tr>
<td>pagenum</td>
<td>Starting page number for a secondary database file</td>
</tr>
<tr>
<td>diff_file</td>
<td>File path and name for DIFFERENCE files (.delta files)</td>
</tr>
</tbody>
</table>

The `CREATE DATABASE` statement creates a new database. You can use `CREATE DATABASE` or `CREATE SCHEMA`. They are synonymous.

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 when 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 the absolute path, when creating either the database or an alias for it.

**Using a Database Alias**

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

```
alias = filepath
```

**Creating a Database Remotely**

If you create a database on a remote server, you should 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:

```
servername[/{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:

```
\servername\{filepath | db_alias}
```

**Optional Parameters for CREATE DATABASE**

**USER and PASSWORD**

Clauses for specifying the user name and the password, respectively, of an existing user in the security database security2.fdb. 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.

**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 changed automatically to the default page size, 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 4,096.

**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 applied to CHAR, VARCHAR and BLOB 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.

**SET SQL 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.

**Examples Using CREATE DATABASE**

1. Creating a database in Windows, located on disk D with a page size of 8,192. The owner of the database will be the user wizard. The database will be in Dialect 1 and it will use WIN1251 as its default character set.

```sql
SET SQL DIALECT 1;
CREATE DATABASE 'D:\test.fdb'
USER 'wizard' PASSWORD 'player'
PAGE_SIZE = 8192 DEFAULT CHARACTER SET WIN1251;
```

2. Creating a database in the Linux operating system with a page size of 4,096. The owner of the database will be the user wizard. The database will be in Dialect 3 and it will use UTF8 as its default character set, with UNICODE_CI_AI as the default collation.

```sql
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 aliases.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'
PAGE_SIZE = 8192
DEFAULT CHARACTER SET UTF8
FILE 'D:\test.fdb2'
STARTING AT PAGE 10001
FILE 'D:\test.fdb3'
STARTING AT PAGE 20001;
```

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.

```
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 or toggling its “copy-safe” state

*Available in*

DSQL — both functions. ESQL — file reorganisation only

**Syntax**

```
ALTER {DATABASE | SCHEMA}
    [<add_sec_clause> [<add_sec_clause> ...]]
    [{ADD DIFFERENCE FILE 'diff_file' | DROP DIFFERENCE FILE}]
    [{BEGIN | END} BACKUP]

<add_sec_clause> ::= ADD <sec_file> [<sec_file> ...]

<sec_file> ::= FILE 'filepath'
    [STARTING [AT [PAGE]] pagenum]
    [LENGTH [=] num [PAGE[s]]]
```

Multiple files can be added in one ADD clause:

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

Multiple ADD FILE clauses are allowed; and 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 17. ALTER DATABASE Statement Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_sec_clause</td>
<td>Adding a secondary database file</td>
</tr>
<tr>
<td>sec_file</td>
<td>File specification for secondary file</td>
</tr>
<tr>
<td>filepath</td>
<td>Full path and file name of the delta file or the secondary database file</td>
</tr>
<tr>
<td>pagenum</td>
<td>Page number from which the secondary database file is to start</td>
</tr>
<tr>
<td>num</td>
<td>Maximum size of the secondary file in pages</td>
</tr>
<tr>
<td>diff_file</td>
<td>File path and name of the .delta file (difference file)</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)

Only administrators have the authority to use ALTER DATABASE.

**Parameters for** ALTER DATABASE

**ADD FILE**

Adds a secondary file 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**

This is the clause that 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**

This is the clause that 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**

This is the clause used to switch 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 to run gbak while the database is in LOCKED or MERGE state.

**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.

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

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

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

3. Deleting the description of the delta file:

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

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

   ```
   ALTER DATABASE
   BEGIN BACKUP;
   ```

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

   ```
   ALTER DATABASE
   END BACKUP;
   ```

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*

```sql
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.

Only administrators have the authority to use `DROP DATABASE`.

*Example*
Deleting the database the client is connected to.

```sql
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**

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

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

<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</td>
</tr>
<tr>
<td></td>
<td>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**
Clause specifying 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 increasing in size as long as it is necessary.

**STARTING AT**
Clause specifying 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.

Only administrators have the authority to use CREATE SHADOW.

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;`

**Examples Using CREATE SHADOW**

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

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

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

   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*

```
DROP SHADOW sh_num
```
Table 19. DROP SHADOW Statement Parameter

<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 database one is connected to. When a shadow is dropped, all files related to it are deleted and shadowing to the specified `sh_num` ceases.

Only administrators have the authority to use DROP SHADOW.

Example of Dropping a Shadow
Deleting “shadow number 1”.

```sql
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> ::=  
 {SMALLINT | INTEGER | BIGINT} [<array_dim>]  
 | {FLOAT | DOUBLE PRECISION} [<array_dim>]
```
Table 20. CREATE DOMAIN Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Domain name consisting of up to 31 characters</td>
</tr>
<tr>
<td><strong>Parameter</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>datatype</td>
<td>SQL data type</td>
</tr>
<tr>
<td>literal</td>
<td>A literal value that is compatible with <code>datatype</code></td>
</tr>
<tr>
<td>context_var</td>
<td>Any context variable whose type is compatible with <code>datatype</code></td>
</tr>
<tr>
<td>dom_condition</td>
<td>Domain condition</td>
</tr>
<tr>
<td>collation_name</td>
<td>Name of a collation sequence that is valid for <code>charset_name</code>, if it is</td>
</tr>
<tr>
<td></td>
<td>supplied with <code>datatype</code> or, otherwise, is valid for the default character set</td>
</tr>
<tr>
<td></td>
<td>of the database</td>
</tr>
<tr>
<td>array_dim</td>
<td>Array dimensions</td>
</tr>
<tr>
<td>m, n</td>
<td>Integer numbers defining the index range of an array dimension</td>
</tr>
<tr>
<td>precision</td>
<td>The total number of significant digits that a value of the <code>datatype</code> can hold</td>
</tr>
<tr>
<td></td>
<td>(1..18)</td>
</tr>
<tr>
<td>scale</td>
<td>The number of digits after the decimal point (0..<code>precision</code>)</td>
</tr>
<tr>
<td>size</td>
<td>The maximum size of a string in characters</td>
</tr>
<tr>
<td>charset_name</td>
<td>The name of a valid character set, if the character set of the domain is to</td>
</tr>
<tr>
<td></td>
<td>be different to the default character set of the database</td>
</tr>
<tr>
<td>subtype_num</td>
<td>BLOB subtype number</td>
</tr>
<tr>
<td>subtype_name</td>
<td>BLOB subtype mnemonic name</td>
</tr>
<tr>
<td>seglen</td>
<td>Segment size (max. 65535)</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</td>
</tr>
<tr>
<td></td>
<td>rows</td>
</tr>
<tr>
<td>expression</td>
<td>An expression resolving to a value that is compatible with <code>datatype</code></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 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.

**CHARACTER 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` in creating 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 BLOBs 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.

Any user connected to the database can create a domain.

**CREATE DOMAIN Examples**

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

```
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.

```
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.

```
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.

```
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.

```
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.

```
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]
    [ADD [CONSTRAINT] CHECK (<dom_condition>) | DROP CONSTRAINT]
```

```
<datatype> ::= {SMALLINT | INTEGER | BIGINT}
| {FLOAT | DOUBLE PRECISION}
| {DATE | TIME | TIMESTAMP}
| {DECIMAL | NUMERIC} [((precision [, scale])]
| {CHAR | CHARACTER} [VARYING] | VARCHAR [((size)]
| CHARACTER SET charset_name
| {NCHAR | NATIONAL {CHARACTER | CHAR}} [VARYING] [((size)]
```

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

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

Table 21. 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>datatype</td>
<td>SQL data type</td>
</tr>
<tr>
<td>literal</td>
<td>A literal value that is compatible with datatype</td>
</tr>
<tr>
<td>context_var</td>
<td>Any context variable whose type is compatible with datatype</td>
</tr>
<tr>
<td>precision</td>
<td>The total number of significant digits that a value of the datatype can hold</td>
</tr>
<tr>
<td></td>
<td>(1..18)</td>
</tr>
<tr>
<td>scale</td>
<td>The number of digits after the decimal point (0..precision)</td>
</tr>
</tbody>
</table>
### Parameter | Description
---|---
size | The maximum size of a string in characters
charset_name | The name of a valid character set, if the character set of the domain is to be changed
subtype_num | BLOB subtype number
subtype_name | BLOB subtype mnemonic name
seglen | Segment size (max. 65535)
select_one | A scalar SELECT statement — selecting one column and returning only one row
select_list | A SELECT statement selecting one column and returning zero or more rows
select_expr | A SELECT statement selecting one or more columns and returning zero or more rows
expression | An expression resolving to a value that is compatible with *datatype*
genname | Sequence (generator) name
func | Internal function or UDF

The `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 `ALTER DOMAIN` statement.

**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`.

**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.

Any user connected to the database can alter a domain, provided it is not prevented by dependencies from objects to which that user does not have sufficient privileges.

**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.
- In Firebird 2.5 and lower, the **NOT NULL** constraint can be neither enabled nor disabled for a domain.
- There is no way to change the default collation without dropping the domain and recreating it with the desired attributes.

**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:
See also

CREATE DOMAIN, DROP DOMAIN

5.3.3. DROP DOMAIN

Used for
Deleting an existing domain

Available in
DSQL, ESQL

Syntax

```sql
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.

Any user connected to the database can drop a domain.

Example

Deleting the COUNTRYNAME domain:

```sql
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**

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

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

<regular_col_def> ::=
  colname {<datatype> | domainname}
  [DEFAULT {<literal> | NULL | <context_var>}]
  [NOT NULL]
  [<col_constraint>]
  [COLLATE collation_name]

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

<datatype> ::= 
  {SMALLINT | INTEGER | BIGINT} [<array_dim>]
  | {FLOAT | DOUBLE PRECISION} [<array_dim>]
  | {DATE | TIME | TIMESTAMP} [<array_dim>]
  | {DECIMAL | NUMERIC} [(precision [, scale]) [<array_dim>]]
  | {CHAR | CHARACTER} [VARYING] | VARCHAR [(size)]
  | [array_dim] [CHARACTER SET charset_name]
  | {NCHAR | NATIONAL {CHARACTER | CHAR}} [VARYING]
  | [(size)] [<array_dim>]
  | BLOB [SUB_TYPE {subtype_num | subtype_name}]
  | [SEGMENT SIZE seglen] [CHARACTER SET charset_name]
  | BLOB [[(seglen [, subtype_num])]]

<array_dim> ::= '[' [m:]n [, [m:]n ... ']'

<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>)
```
Table 22. 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>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 <code>other_table</code> 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>collation</td>
<td>Collation</td>
</tr>
<tr>
<td>array_dim</td>
<td>Array dimensions</td>
</tr>
<tr>
<td>m, n</td>
<td>Integer numbers defining the index range of an array dimension</td>
</tr>
<tr>
<td>precision</td>
<td>The total number of significant digits that a value of the datatype can hold (1..18)</td>
</tr>
<tr>
<td>scale</td>
<td>The number of digits after the decimal point (0..precision)</td>
</tr>
<tr>
<td>size</td>
<td>The maximum size of a string in characters</td>
</tr>
<tr>
<td>charset_name</td>
<td>The name of a valid character set, if the character set of the column is to be different to the default character set of the database</td>
</tr>
<tr>
<td>subtype_num</td>
<td>BLOB subtype number</td>
</tr>
<tr>
<td>subtype_name</td>
<td>BLOB subtype mnemonic name</td>
</tr>
<tr>
<td>seglen</td>
<td>Segment size (max. 65535)</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>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</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.

**Making a Column Non-nullable**

In Firebird, columns are nullable by default. The optional `NOT NULL` clause specifies that the column cannot take `NULL` in place of a value.

**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 character set specified during the creation of the database will be used by default. If no character set was specified during the creation of the database, the `NONE` character set is applied by default. 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 collation sequence that is default for the specified character set during the creation of the column is applied by default.

**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 to make the domain nullable and apply NOT NULL in the downstream column definitions and variable declarations.

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**

Four types of constraints can be specified. They are:

- Primary key (`PRIMARY KEY`)
- Unique key (`UNIQUE`)
- Foreign key (`REFERENCES`)
- CHECK constraint (`CHECK`)

Constraints can be specified at column level (“column constraints”) or at table level (“table constraints”). Table-level constraints are needed when keys (uniqueness constraint, Primary Key, Foreign Key) are to be formed across multiple columns and when a CHECK constraint involves other columns in the row besides the column being defined. Syntax for some types of constraint may differ slightly according to whether the constraint is being defined at 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.
- Table-level constraints are specified after all of the column definitions. They 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. 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.

**PRIMARY KEY**

The PRIMARY KEY constraint is built on one or more key columns, each column having the NOT NULL constraint specified for it. 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.
## FOREIGN KEY

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

```
... 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.

CHECK conditions — whether defined at table level or column level — refer to table columns by their names. The use of the keyword VALUE as a placeholder, as in domain CHECK constraints, is not valid in the context of defining column constraints.

Example

with two column-level constraints and one at table-level:
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)  
);  

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

```
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

Restrictions on GTTs

GT Ts 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:

```sql
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:

```sql
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.

**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. It creates a security vulnerability and is not recommended.

**External File Format**

The “row” format of the external table is fixed length. 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 will appear to external applications as unparseable “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):

```sql
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
```
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],
    PRIMARY KEY (JOB_CODE, JOB_GRADE),
    FOREIGN KEY (JOB_COUNTRY) REFERENCES COUNTRY (COUNTRY)
    ON UPDATE CASCADE
    ON DELETE SET NULL,
    CONSTRAINT CHK_SALARY CHECK (MIN_SALARY < MAX_SALARY)
);
```

4. Creating the PROJECT table with primary, foreign and unique key constraints with custom index names specified with the USING clause.
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,  
    CONSTRAINT PK_PROJECT PRIMARY KEY (PROJ_ID) USING INDEX IDX_PROJ_ID,  
    FOREIGN KEY (TEAM_LEADER) REFERENCES EMPLOYEE (EMP_NO)  
        USING INDEX IDX_LEADER);  

5. 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,  
    UPDATER_ID VARCHAR(20) NOT NULL,  
    OLD_SALARY SALARY NOT NULL,  
    PERCENT_CHANGE DOUBLE PRECISION DEFAULT 0 NOT NULL,  
    SALARY_CHANGE GENERATED ALWAYS AS  
        (OLD_SALARY * PERCENT_CHANGE / 100),  
    NEW_SALARY COMPUTED BY  
        (OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100)  
);  
```


```sql  
CREATE GLOBAL TEMPORARY TABLE MYCONNGTT (  
    ID INTEGER NOT NULL PRIMARY KEY,  
    TXT VARCHAR(32),  
    TS TIMESTAMP DEFAULT CURRENT_TIMESTAMP)  
ON COMMIT PRESERVE ROWS;  
```

7. 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.

```sql  
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;  
```
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>

<regular_col_def> ::= 
    colname {<datatype> | domainname}
    [DEFAULT {<literal> | NULL | <context_var>}] 
    [NOT NULL]
    [<col_constraint>]
    [COLLATE collation_name]

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

<col_mod> ::= <regular_col_mod> | <computed_col_mod>

<regular_col_mod> ::= 
    TO  newname
| POSITION newpos
| TYPE {<datatype> | domainname}
| SET DEFAULT {<literal> | NULL | <context_var>}
| DROP DEFAULT

<computed_col_mod> ::= 
    TO  newname
| POSITION newpos
| [TYPE <datatype>] {COMPUTED [BY] | GENERATED ALWAYS AS} (<expression>)

<datatype> ::= 
    {SMALLINT | INTEGER | BIGINT} [array_dim]
| {FLOAT | DOUBLE PRECISION} [array_dim]
| {DATE | TIME | TIMESTAMP} [array_dim]
```
Chapter 5. Data Definition (DDL) Statements

| {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>] |
| {CHAR | CHARACTER} [VARYING] | VARCHAR} [(size)] |
| [array_dim] | [CHARACTER SET charset_name] |
| {NCHAR | NATIONAL {CHARACTER | CHAR}} [VARYING] [(size)] [<array_dim>] |
| BLOB [SUB_TYPE {subtype_num | subtype_name}] |
| [SEGMENT SIZE seglen] [CHARACTER SET charset_name] |
| BLOB [(seglen [, subtype_num])] |

<array_dim> ::= ['[ ' [m:]n ,[m:]n ... ']' | ' ']
<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>) } |
	<col_list> ::= colname [, colname ...] |
<using_index> ::= USING |
[ASCENDING] | DESCENDING] INDEX indexname |
<check_condition> ::= |
<val> <operator> <val> |
| <val> [NOT] BETWEEN <val> AND <val> |
| <val> [NOT] IN (|, <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>) |
| <search_condition> |
| NOT <search_condition> |
| <search_condition> OR <search_condition> |
### Table 23. 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 unique in the table.</td>
</tr>
<tr>
<td>newname</td>
<td>New name (identifier) for the column, max. 31 characters. Must be 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 the table)</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>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 TRUE or UNKNOWN/NUL</td>
</tr>
<tr>
<td>collation</td>
<td>Name of a collation sequence that is valid for charset_name, if it is supplied with datatype or, otherwise, is valid for the default character set of the database</td>
</tr>
<tr>
<td>array_dim</td>
<td>Array dimensions</td>
</tr>
<tr>
<td>m, n</td>
<td>Integer numbers defining the index range of an array dimension</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>precision</td>
<td>The total number of significant digits that a value of the <em>datatype</em> can hold (1..18)</td>
</tr>
<tr>
<td>scale</td>
<td>The number of digits after the decimal point (0..<em>precision</em>)</td>
</tr>
<tr>
<td>size</td>
<td>The maximum size of a string in characters</td>
</tr>
<tr>
<td>charset_name</td>
<td>The name of a valid character set, if the character set of the column is to be different from the default character set of the database</td>
</tr>
<tr>
<td>subtype_num</td>
<td>BLOB subtype number</td>
</tr>
<tr>
<td>subtype_name</td>
<td>BLOB subtype mnemonic name</td>
</tr>
<tr>
<td>seglen</td>
<td>Segment size (max. 65535)</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 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 `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 should 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. Be careful about adding a new column with the NOT NULL constraint set. It may lead to breaking the logical integrity of data, since you will have existing records containing NULL in a non-nullable column. When adding a non-nullable column, it is recommended either to set a default value for it or to update the column in existing rows with a non-null value.

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.

The DROP Clause

The DROP <column name> 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)

Renaming a Column: the \texttt{TO} Keyword

The \texttt{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: \texttt{PRIMARY KEY}, \texttt{UNIQUE key}, \texttt{FOREIGN KEY}, column constraint or the \texttt{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 \texttt{TYPE} Keyword

The \texttt{TYPE} keyword 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 \texttt{CHAR} or \texttt{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 \texttt{POSITION} Keyword

The \texttt{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 \texttt{DROP DEFAULT} and \texttt{SET DEFAULT} Clauses

The optional \texttt{DROP DEFAULT} clause deletes the default value for the column if it was put there previously by a \texttt{CREATE TABLE} or \texttt{ALTER TABLE} statement.

• If the column is based on a domain with a default value, the default value will revert to the
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 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.

**Attributes that Cannot Be Altered**

The following alterations are not supported:

- Enabling or disabling the NOT NULL constraint on a column
- Changing the default collation for a character type column

Only the table owner and administrators have the authority to use ALTER TABLE.

**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 UNIQUE constraint and deleting the CURRENCY column.

   ```sql
   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.

   ```sql
   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. 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
deleting a table

Available in
DSQL, ESQL

Syntax

DROP TABLE tablename

Table 24. 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 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 triggers for its events and indexes built for its fields will be deleted as well.

Only the table owner and administrators have the authority to use DROP TABLE.

Example
Deleting 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
Creating or recreating the COUNTRY table.

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 of 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] [ASCENDING | DESCENDING]
INDEX indexname ON tablename
{(col [, col …]) | COMPUTED BY (<expression>)}
```

<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</td>
</tr>
<tr>
<td></td>
<td>computed fields cannot be used in an index</td>
</tr>
<tr>
<td>expression</td>
<td>The expression that will compute the values for a computed index, also</td>
</tr>
<tr>
<td></td>
<td>known as an “expression 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:

```sql
ALTER TABLE COUNTRY ADD CONSTRAINT PK_COUNTRY
    PRIMARY KEY (ID);
```

**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 \textit{compound}, or multi-column, or multi-segment) indexes. Duplicated \texttt{NULL}s are permitted, in accordance with the SQL:99 standard, in both single-segment and multi-segment indexes.

\textbf{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 \texttt{ASC\{ENDING\}} and \texttt{DESC\{ENDING\}} are used to specify the direction of the index. The default index order is \texttt{ASC\{ENDING\}}. It is quite valid to define both an ascending and a descending index on the same column or key set.

\begin{verse}
A descending index can be useful on a column that will be subjected to searches on the high values ("newest", maximum, etc.)
\end{verse}

\textbf{Computed (Expression) Indexes}

In creating an index, you can use the \texttt{COMPUTED \ BY} clause to specify an expression instead of one or more columns. Computed indexes are used in queries where the condition in a \texttt{WHERE}, \texttt{ORDER \ BY} or \texttt{GROUP \ BY} clause exactly matches the expression in the index definition. The expression in a computed index may involve several columns in the table.

\begin{verse}
You can actually create a computed index on a computed field, but the index will never be used.
\end{verse}

\textbf{Limits on Indexes}

Certain limits apply to indexes.

The maximum length of a key in an index is limited to ¼ of the page size.

\textbf{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.

\textit{Table 26. Maximum Indexes per Table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Page Size} & \textbf{Number of Indexes Depending on Column Count} & & \\
& \textbf{Single} & \textbf{2-Column} & \textbf{3-Column} \\
\hline
4096 & 203 & 145 & 113 \\
8192 & 408 & 291 & 227 \\
16384 & 818 & 584 & 454 \\
\hline
\end{tabular}
\end{table}

\textbf{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 27. Maximum indexable (VAR)CHAR length**

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Maximum Indexable String Length by Charset Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 byte/char</td>
</tr>
<tr>
<td>4096</td>
<td>1015</td>
</tr>
<tr>
<td>8192</td>
<td>2039</td>
</tr>
<tr>
<td>16384</td>
<td>4087</td>
</tr>
</tbody>
</table>

Only the table owner and administrators have the authority to use `CREATE INDEX`.

**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:
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 28. 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.

- 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 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.

- 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 housekeeping to do, occasionally, on the indexes of a large table in a database that has frequent inserts, updates or deletes but is infrequently restored.

Use of ALTER INDEX on a Constraint Index

Altering the enforcing 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.

Only the table owner and administrators have the authority to use ALTER INDEX.

ALTER INDEX Examples

1. Deactivating the IDX_UPDATER index

   ALTER INDEX IDX_UPDATER INACTIVE;

2. Switching the IDX_UPDATER index back to the active state and rebuilding it

   ALTER INDEX IDX_UPDATER ACTIVE;

See also

CREATE INDEX, DROP INDEX, SET STATISTICS

5.5.3. DROP INDEX

Used for
Deleting an index

Available in
DSQL, ESQL

Syntax

DROP INDEX indexname

Table 29. 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 deletes the named index from the database.

A constraint index cannot be deleted using DROP INDEX. Constraint indexes are dropped during the process of executing the command ALTER TABLE ... DROP CONSTRAINT ....

Only the table owner and administrators have the authority to use DROP INDEX.

**DROP INDEX Example**

Deleting 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*

```sql
SET STATISTICS INDEX indexname
```

*Table 30. 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.

**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.

The selectivity of an index can be recalculated by the owner of the table or an administrator. 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 ]

<full_column_list> ::= ( colname [, colname ...] )
```

*Table 31. CREATE 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>colname</td>
<td>View column name. Duplicate column names are not allowed.</td>
</tr>
</tbody>
</table>

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 this 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 updatable 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 is covered by 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, such as MIN, MAX, AVG, SUM, COUNT, LIST
- the SELECT statement contains no ORDER BY or GROUP BY clause
- the SELECT statement does not include the keyword DISTINCT or row-restrictive keywords such as ROWS, FIRST, SKIP

**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.

**Ownership of a View**

The creator of a view becomes its owner.

To create a view, a non-admin user 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 base object(s).

Granting other users privileges on the view is only possible if the view owner himself 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).

   ```sql
   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>
<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.
Only the view owner and administrators have the authority to use ALTER VIEW.

**Example using ALTER VIEW**

Altering the view PRICE_WITH_MARKUP

```sql
ALTER VIEW PRICE_WITH_MARKUP (
    CODE_PRICE,
    COST,
    COST * 1.15
) 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*

```sql
CREATE OR ALTER VIEW viewname [<full_column_list>]
AS <select_statement>
[WITH CHECK OPTION]

<full_column_list> ::= (colname [, colname ...])
```

*Table 33. 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
Creating the new view PRICE_WITH_MARKUP view or altering it if it already exists:

```
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

```
DROP VIEW viewname
```

Table 34. 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 deletes an existing view. The statement will fail if the view has dependencies.

Only the view owner and administrators have the authority to use DROP VIEW.

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 35. 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>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**
Creating the new view `PRICE_WITH_MARKUP` view or recreating it, if it already exists.

```
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 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.

Another form of trigger — known as a “database trigger” — can be specified to fire in association with the start or end of a user session (connection) or a user transaction.

5.7.1. CREATE TRIGGER

*Used for*
Creating a new trigger

*Available in*
DSQL, ESQL
Syntax

```sql
CREATE TRIGGER trigname
  { <relation_trigger_legacy>
  | <relation_trigger_sql2003>
  | <database_trigger> }
AS
  [<declarations>]
BEGIN
  [<PSQL_statements>]
END

<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]

<mutation_list> ::= 
  <mutation> [OR <mutation> [OR <mutation>]]

<mutation> ::= { INSERT | UPDATE | DELETE }

<db_event> ::= 
  { CONNECT
  | DISCONNECT
  | TRANSACTION START
  | TRANSACTION COMMIT
  | TRANSACTION ROLLBACK }

<declarations> ::= {<declare_var> | <declare_cursor>};
  [{<declare_var> | <declare_cursor>}; ...]
```

Table 36. 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>
</tbody>
</table>
### Parameter | Description
--- | ---
relation_trigger_sql2003 | Relation trigger declaration compliant with the SQL:2003 standard

database_trigger | Database trigger declaration

tablename | Name of the table with which the relation trigger is associated

viewname | Name of the view with which the relation trigger is associated

mutation_list | List of relation (table | view) events

number | Position of the trigger in the firing order. From 0 to 32,767

db_event | Connection or transaction event

declarations | Section for declaring local variables and named cursors

declare_var | Local variable declaration

declare_cursor | Named cursor declaration

PSQL_statements | Statements in Firebird’s programming language (PSQL)

---

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), or for a `database 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 relation trigger), the phase of the trigger and the event[s] it applies to. The 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`.

### Relation Triggers (on Tables or Views)

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.

### 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.
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.

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 file 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

**Execution of Database Triggers and Exception Handling**

CONNECT and DISCONNECT triggers are executed in a transaction created specifically for this purpose. 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.

**Trigger Suppression**

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.\n';

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
```

See also
ALTER TRIGGER, CREATE OR ALTER TRIGGER, RECREATE TRIGGER, DROP TRIGGER

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> | ON <db_event>]
[POSITION number]
[
   AS
   [<declarations>]
   BEGIN
   [<PSQL_statements>]
   END
]
```

Table 37. ALTER TRIGGER Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trigname</td>
<td>Name of an existing trigger</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>declarations</td>
<td>Section for declaring local variables and named cursors</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>PSQL_statements</td>
<td>Statements in Firebird’s programming language (PSQL)</td>
</tr>
</tbody>
</table>

The ALTER TRIGGER statement allows certain changes to the header and body of a trigger.
Permitted Changes to Triggers

- Status (ACTIVE | INACTIVE)
- Phase (BEFORE | AFTER)
- Events; but relation trigger events cannot be changed to database trigger events, nor vice versa
- Position in the firing order
- Modifications to code in the trigger body

If some element was not specified, it remains unchanged.

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 relation 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.

Administrators and the following users have the authority to use ALTER TRIGGER:

- For relation triggers, the owner of the table
- For database triggers, the owner of the database

Examples using ALTER TRIGGER

1. Deactivating the set_cust_no trigger (switching it to the inactive status).

   ALTER TRIGGER set_cust_no INACTIVE;

2. Changing the firing order position of the set_cust_no trigger.

   ALTER TRIGGER set_cust_no POSITION 14;

3. Switching the TR_CUST_LOG trigger to the inactive status and modifying the list of events.

   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.
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 trigname { 
  <relation_trigger_legacy> | 
  <relation_trigger_sql2003> | 
  <database_trigger> } 
AS 
  [<declarations>] 
BEGIN 
  [<PSQL_statements>] 
END
```

For the full detail of the syntax, see CREATE TRIGGER.

The CREATE OR ALTER TRIGGER statement creates a new trigger if it does not exist; otherwise it alters and recompiles it with the privileges intact and dependencies unaffected.

Example using 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);
  END

See also
CREATE TRIGGER, ALTER TRIGGER, RECREATE TRIGGER

5.7.4. DROP TRIGGER

Used for
Deleting an existing trigger

Available in
DSQL, ESQL

Syntax

DROP TRIGGER trigneme

Table 38. DROP TRIGGER Statement Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trigneme</td>
<td>Trigger name</td>
</tr>
</tbody>
</table>

The DROP TRIGGER statement deletes an existing trigger.

Administrators and the following users have the authority to use DROP TRIGGER:

- For relation triggers, the owner of the table
- For database triggers, the owner of the database

Example using DROP TRIGGER
Deleting the set_cust_no trigger.

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 trigname {
  <relation_trigger_legacy> | <relation_trigger_sql2003> | <database_trigger> }
AS
  [<declarations>]
BEGIN
  [<PSQL_statements>]
END
```

For the full detail of the syntax, see CREATE TRIGGER.

The RECREATE TRIGGER statement creates a new trigger if no trigger with the specified name exists; otherwise the RECREATE TRIGGER statement tries to delete 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 using 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, an executable block or a trigger. Stored procedures, executable blocks and triggers are written in procedural SQL (PSQL). Most SQL statements are available in PSQL as well, sometimes with limitations or extensions. Among notable exceptions are DDL and transaction control statements.
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 proname
  ([<inparam> [, <inparam> ...]])
  [RETURNS (<outparam> [, <outparam> ...])]
AS
  [<declarations>]
BEGIN
  [<PSQL_statements>]
END

<inparam> ::= <param_decl> [{= | DEFAULT} <value>]

<outparam> ::= <param_decl>

<value> ::= {<literal> | NULL | <context_var>}

<param_decl> ::= paramname <type> [NOT NULL]
  [COLLATE collation]

<type> ::= 
  <datatype>
  | [TYPE OF] domain
  | TYPE OF COLUMN rel.col

<datatype> ::= 
  {SMALLINT | INT[GER] | BIGINT}
  | {FLOAT | DOUBLE PRECISION}
  | {DATE | TIME | TIMESTAMP}
  | {DECIMAL | NUMERIC} [(precision [, scale])]
  | {CHAR | CHARACTER} [VARYING] | VARCHAR [(size)]
  | [CHARACTER SET charset]
  | {NCHAR | NATIONAL {CHARACTER | CHAR}} [VARYING]
  | [(size)]
  | BLOB [SUB_TYPE {subtype_num | subtype_name}]
  | [SEGMENT SIZE seglen] [CHARACTER SET charset]
  | BLOB [(seglen [, subtype_num])]`
```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>proname</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>
<tr>
<td>declarations</td>
<td>Section for declaring local variables and named cursors</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>PSQL_statements</td>
<td>Procedural SQL statements</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 whose type is compatible with the data type of the parameter</td>
</tr>
<tr>
<td>paramname</td>
<td>The name of an input or output parameter of the 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 and its local variables</td>
</tr>
<tr>
<td>datatype</td>
<td>SQL data type</td>
</tr>
<tr>
<td>collation</td>
<td>Collation sequence</td>
</tr>
<tr>
<td>domain</td>
<td>Domain name</td>
</tr>
<tr>
<td>rel</td>
<td>Table or view name</td>
</tr>
<tr>
<td>col</td>
<td>Table or view column name</td>
</tr>
<tr>
<td>precision</td>
<td>The total number of significant digits that the parameter should be able to hold (1..18)</td>
</tr>
<tr>
<td>scale</td>
<td>The number of digits after the decimal point (0..precision)</td>
</tr>
<tr>
<td>size</td>
<td>The maximum size of a string type parameter or variable, in characters</td>
</tr>
<tr>
<td>charset</td>
<td>Character set of a string type parameter or variable</td>
</tr>
<tr>
<td>subtype_num</td>
<td>BLOB subtype number</td>
</tr>
<tr>
<td>subtype_name</td>
<td>BLOB subtype mnemonic name</td>
</tr>
<tr>
<td>seglen</td>
<td>Segment size (max. 65535)</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 specified for it. 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 procedure. They are passed into the procedure as values, so anything that changes them inside the procedure has no effect on the parameters in the calling program. Input parameters may have default values. Those that do have values specified for them must be located 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.

**Use of Domains in Declarations**

A domain name can be specified as the type of a parameter. The parameter will inherit all domain attributes. If a default value is specified for the parameter, 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 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 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.
Bug warning for pre-Firebird 3 versions:

For input parameters, the collation that comes with the column’s type is ignored in comparisons (e.g. equality tests). For local variables, the behaviour varies.

The bug was fixed for Firebird 3.

Variable and Cursor Declarations

The optional declarations section, located last in the header section of the procedure definition, defines variables local to the procedure and its named cursors. Local variable declarations follow the same rules as parameters regarding specification of the data type. See details in the PSQL chapter for DECLARE VARIABLE and DECLARE CURSOR.

Procedure Body

The header section is followed by the procedure 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.

Any user connected to the database can create a new stored procedure. The user who creates a stored procedure becomes its owner.

Examples

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;  
END
```

Creating a selectable stored procedure that generates data for mailing labels (from employee.fdb):
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
    INTO :customer, :first_name, :last_name, :addr1, :addr2, 
        :city, :state, :country, :postcode;

    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 = state || ' ' || postcode;
            END IF;
        END IF;
    ELSE
        BEGIN
            IF (city IS NOT NULL) THEN
                line5 = city || ', ' || state;
            ELSE
                line5 = state;
            END IF;
        END IF;
    END IF;
END;
ELSE
    line5 = state;
    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
  [(<inparam> [, <inparam> ...])]
  [RETURNS (<outparam> [, <outparam> ...])]
AS
  [declare_var]
BEGIN
  [PSQL_statements]
END

<inparam> ::= <param_decl> [= | DEFAULT] <value>

<outparam> ::= <param_decl>

<param_decl> ::= paramname <type> [NOT NULL]
  [COLLATE collation]

<type> ::=<datatype>
  | [TYPE OF] domain
  | TYPE OF COLUMN rel.col
<datatype> ::= {SMALLINT | INT[GER] | BIGINT}
  | {FLOAT | DOUBLE PRECISION}
  | {DATE | TIME | TIMESTAMP}
  | {DECIMAL | NUMERIC} [(precision [, scale])]
  | {CHAR | CHARACTER} [VARYING] | VARCHAR [[size]]
    [CHARACTER SET charset]
  | {NCHAR | NATIONAL {CHARACTER | CHAR} [VARYING]
    [[size]]
  | BLOB [SUB_TYPE {subtype_num | subtype_name}]
    [SEGMENT SIZE seglen] [CHARACTER SET charset]
  | BLOB [[seglen [, subtype_num]]]
<declarations> ::= {declare_var} | declare_cursor;
  [{declare_var} | declare_cursor]; ...
```

### Table 40. ALTER PROCEDURE Statement Parameters

<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>
<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>Section for declaring local variables and named cursors</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>PSQL_statements</td>
<td>Procedural SQL statements</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 whose type is compatible with the data type of the parameter</td>
</tr>
<tr>
<td>paramname</td>
<td>The name of an input or output parameter of the 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 and its local variables</td>
</tr>
<tr>
<td>datatype</td>
<td>SQL data type</td>
</tr>
<tr>
<td>collation</td>
<td>Collation sequence</td>
</tr>
<tr>
<td>domain</td>
<td>Domain name</td>
</tr>
<tr>
<td>rel</td>
<td>Table or view name</td>
</tr>
<tr>
<td>col</td>
<td>Table or view column name</td>
</tr>
<tr>
<td>precision</td>
<td>The total number of significant digits that the parameter should be able to hold (1..18)</td>
</tr>
<tr>
<td>scale</td>
<td>The number of digits after the decimal point (0..precision)</td>
</tr>
<tr>
<td>size</td>
<td>The maximum size of a string type parameter or variable, in characters</td>
</tr>
<tr>
<td>charset</td>
<td>Character set of a string type parameter or variable</td>
</tr>
<tr>
<td>subtype_num</td>
<td>BLOB subtype number</td>
</tr>
<tr>
<td>subtype_name</td>
<td>BLOB subtype mnemonic name</td>
</tr>
<tr>
<td>seglen</td>
<td>Segment size (max. 65535)</td>
</tr>
</tbody>
</table>

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.

The procedure owner and **Administrators** have the authority to use `ALTER PROCEDURE`.

**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

```
CREATE OR ALTER PROCEDURE proname
    [(<inparam> [, <inparam> ...])]
    [RETURNS (<outparam> [, <outparam> ...])]  
AS
    [declarations]
BEGIN
    [PSQL_statements]
END
```

For the full syntax detail, see CREATE PROCEDURE.

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.
Example

Creating or altering the GET_EMP_PROJ procedure.

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

DROP PROCEDURE procname

Table 41. DROP PROCEDURE Statement Parameter

<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.

The procedure owner and Administrators have the authority to use 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
  [([inparam> [, inparam> ...]])
  [RETURNS (<outparam> [, <outparam> ...])]
  AS
  [declarations>
BEGIN
  [PSQL_statements>
END

For the full syntax detail, see CREATE PROCEDURE.

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.

Example
Creating the new GET_EMP_PROJ stored procedure or recreating the existing GET_EMP_PROJ stored procedure.
RECREATE 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, DROP PROCEDURE, CREATE OR ALTER PROCEDURE

5.9. EXTERNAL FUNCTION

REVIEW STATUS
All sections from this point forward to the end of the chapter are awaiting technical and editorial review.

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 to a database, they become available in dynamic and procedural statements as though they were implemented in the SQL language internally.

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 may be contained in more than one library — or “module”, as it is referred to in the syntax.

5.9.1. DECLARE EXTERNAL FUNCTION

Used for
Declaring a user-defined function (UDF) to the database

Available in
DSQL, ESQL
Table 42. 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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>extension.</td>
</tr>
<tr>
<td>sqltype</td>
<td>SQL data type. It cannot be an array or an array element</td>
</tr>
<tr>
<td>length</td>
<td>The maximum length of a null-terminated string, specified in bytes</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</td>
</tr>
<tr>
<td></td>
<td>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. As well as the SQL types, the CSTRING type is available for specifying a null-terminated string with a maximum length of LENGTH bytes.

By default, input parameters are passed by reference. The BY_DESCRIPTOR clause may be specified
instead, if the input parameter is passed by descriptor. Passing a parameter by descriptor makes it possible to process NULLs.

**Clauses and Keywords**

**RETURNS clause**

(Required) specifies the output parameter returned by the function. A function is scalar: it returns one and only one 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).

**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.

```sql
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.

```sql
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.

```sql
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.

```sql
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`

### 5.9.2. 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 43. ALTER EXTERNAL FUNCTION Statement Parameters**
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.

**Examples using ALTER EXTERNAL FUNCTION**

1. Changing the entry point for an external function

   ```sql
   ALTER EXTERNAL FUNCTION invl ENTRY_POINT 'intNvl';
   ```

2. Changing the module name for an external function

   ```sql
   ALTER EXTERNAL FUNCTION invl MODULE_NAME 'fbudf2';
   ```

**See also**

DECLARE EXTERNAL FUNCTION, DROP EXTERNAL FUNCTION

### 5.9.3. 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 44. 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.

*Example using DROP EXTERNAL FUNCTION*

Deleting the declaration of the `addDay` function.

```sql
DROP EXTERNAL FUNCTION addDay;
```

*See also*

`DECLARE EXTERNAL FUNCTION`

### 5.10. FILTER

A `BLOB FILTER` filter is a database object that is actually 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](#).

#### 5.10.1. 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'

<sub_type> ::= number | <mnemonic>

<mnemonic> ::= |
   | BINARY | TEXT | BLR | ACL | RANGES
   | SUMMARY | FORMAT | TRANSACTION_DESCRIPTION
   | EXTERNAL_FILE_DESCRIPTION | user_defined
```

Table 45. DECLARE FILTER Statement Parameters

<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</td>
</tr>
<tr>
<td></td>
<td>not be the same name as the name exported from the filter library via</td>
</tr>
<tr>
<td></td>
<td>ENTRY_POINT.</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:

```sql
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 mnemonic names in upper case, they can be used case-insensitively and without quotation marks when a filter is declared.

**Warning**

From Firebird 3 onward, the system tables will no longer be writable by users. However, inserting custom types into `RDB$TYPES` is still possible.

**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.

**Examples of FILTER**

1. Creating a BLOB filter using subtype numbers.

```sql
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.

```sql
DECLARE FILTER FUNNEL
    INPUT_TYPE blr OUTPUT_TYPE text
    ENTRY_POINT 'blr2asc' MODULE_NAME 'myfilterlib';
```

*See also*
DROP FILTER

5.10.2. DROP FILTER

Used for
Removing a BLOB filter declaration from the database

Available in
DSQL, ESQL

Syntax

```
DROP FILTER filtername
```

<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.

Any user connected to the database can drop a BLOB filter.

Example
Deleting a BLOB filter.

```
DROP FILTER DESC_FILTER;
```

See also
DECLARE FILTER

5.11. 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”. Both terms are implemented in Firebird, which recognises and 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 cutting the values, which will compromise the uniqueness of the series.

This section describes how to create, set and delete sequences.

### 5.11.1. CREATE SEQUENCE (GENERATOR)

**Used for**

Creating a new SEQUENCE (GENERATOR)

**Available in**

DSQL, ESQL

**Syntax**

```sql
CREATE {SEQUENCE | GENERATOR} seq_name
```

**Table 47. CREATE SEQUENCE | CREATE GENERATOR Statement Parameter**

<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>
</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 if standards-compliant metadata management is important.

When a sequence is created, its value is set to 0. Each time the `NEXT VALUE FOR seq_name` operator is used with that sequence, its value increases by 1. The `GEN_ID(seq_name, <step>)` function can be called instead, to “step” the series by a different integer number.

Any user connected to the database can create a sequence (generator).

**Examples**

1. Creating the EMP_NO_GEN series using CREATE SEQUENCE.

   ```sql
   CREATE SEQUENCE EMP_NO_GEN;
   ```

2. Creating the EMP_NO_GEN series using CREATE GENERATOR.

   ```sql
   CREATE GENERATOR EMP_NO_GEN;
   ```

**See also**
### 5.11.2. ALTER SEQUENCE

**Used for**
Setting the value of a sequence or generator to a specified value

**Available in**
DSQL

**Syntax**

```
ALTER SEQUENCE seq_name RESTART WITH new_val
```

<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>new_val</td>
<td>New sequence (generator) value. A 64-bit integer from $-2^{63}$ to $2^{63}-1$.</td>
</tr>
</tbody>
</table>

The `ALTER SEQUENCE` statement sets the current value of a sequence or generator to the specified value.

⚠️ 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.

Any user connected to the database can set the sequence (generator) value.

**Examples**

1. Setting the value of the `EMP_NO_GEN` sequence to 145.

   ```sql
   ALTER SEQUENCE EMP_NO_GEN RESTART WITH 145;
   ```

2. Doing the same thing, using `SET GENERATOR`:

   ```sql
   SET GENERATOR EMP_NO_GEN TO 145;
   ```

**See also**

- `SET GENERATOR`, `CREATE SEQUENCE (GENERATOR)`, `DROP SEQUENCE (GENERATOR)`, `NEXT VALUE FOR`, `GEN_ID()` function

### 5.11.3. SET GENERATOR

**Used for**
Setting the value of a sequence or generator to a specified value
Available in
DSQL, ESQL

Syntax

```plaintext
SET GENERATOR seq_name TO new_val
```

Table 49. 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 SET GENERATOR statement sets the current value of a sequence or generator to the specified value.

![Information icon]

Although SET GENERATOR is considered outdated, it is retained for backward compatibility. Using the standards-compliant ALTER SEQUENCE is current and is recommended.

Any user connected to the database can set the sequence (generator) value.

Examples

1. Setting the value of the EMP_NO_GEN sequence to 145:

   ```sql
   SET GENERATOR EMP_NO_GEN TO 145;
   ```

2. Doing the same thing, using ALTER SEQUENCE:

   ```sql
   ALTER SEQUENCE EMP_NO_GEN RESTART WITH 145;
   ```

See also

ALTER SEQUENCE, CREATE SEQUENCE (GENERATOR)

5.11.4. DROP SEQUENCE (GENERATOR)

Used for
Deleting SEQUENCE (GENERATOR)

Available in
DSQL, ESQL

Syntax

```plaintext
DROP {SEQUENCE | GENERATOR} seq_name
```
The statements DROP SEQUENCE and DROP GENERATOR statements are equivalent: both delete an existing sequence (generator). Either is valid but DROP SEQUENCE, being current, is recommended.

The statements will fail if the sequence (generator) has dependencies.

Any user connected to the database can drop a sequence (generator).

Example
Dropping the EMP_NO_GEN series:

```sql
DROP SEQUENCE EMP_NO_GEN;
```

See also
CREATE SEQUENCE (GENERATOR), ALTER SEQUENCE, SET GENERATOR

### 5.12. EXCEPTION

This section describes how to create, modify and delete custom exceptions for use in error handlers in PSQL modules.

#### 5.12.1. CREATE EXCEPTION

*Used for*
Creating a new exception for use in PSQL modules

*Available in*
DSQL, ESQL

*Syntax*

```sql
CREATE EXCEPTION exception_name 'message'
```

<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>
</tbody>
</table>

The statement CREATE EXCEPTION creates a new exception for use in PSQL modules. If an exception of 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.

Any user connected to the database can create an exception.

Examples
1. Creating an exception named E_LARGE_VALUE:

   ```sql
   CREATE EXCEPTION E_LARGE_VALUE
   'The value is out of range';
   ```

2. Creating an exception named ERROR_REFIN_RATE:

   ```sql
   CREATE EXCEPTION ERROR_REFIN_RATE
   'Error detected in the spread of discount rates';
   ```

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.

Custom exceptions are stored in the system table RDB$EXCEPTIONS.

See also
ALTER EXCEPTION, CREATE OR ALTER EXCEPTION, DROP EXCEPTION, RECREATE EXCEPTION

5.12.2. ALTER EXCEPTION

Used for
Modifying the message returned from a custom exception

Available in
DSQL, ESQL

Syntax

```sql
ALTER EXCEPTION exception_name 'message'
```

<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 ALTER EXCEPTION can be used at any time, to modify the default text of the message. Any user connected to the database can alter an exception message.

**Examples**

1. Changing the default message for the exception E_LARGE_VALUE:

   ```sql
   ALTER EXCEPTION E_LARGE_VALUE
   'The value exceeds the prescribed limit of 32,765 bytes';
   ```

2. Changing the default message for the exception ERROR_REFIN_RATE:

   ```sql
   ALTER EXCEPTION ERROR_REFIN_RATE
   'Rate is outside the allowed range';
   ```

**See also**

CREATE EXCEPTION, CREATE OR ALTER EXCEPTION, DROP EXCEPTION, RECREATE EXCEPTION

### 5.12.3. 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'
```

**Table 53. CREATE OR ALTER EXCEPTION Statement Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception_name</td>
<td>Exception name</td>
</tr>
<tr>
<td>message</td>
<td>Error message. The maximum length is limited to 1,021 characters</td>
</tr>
</tbody>
</table>

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.

Any user connected to the database can use this statement to create an exception or alter the text of one that already exists.
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

5.12.4. DROP EXCEPTION

Used for
Deleting a custom exception

Available in
DSQL, ESQL

Syntax

DROP EXCEPTION exception_name

Table 54. 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.

If an exception is used only in stored procedures, it can be deleted at any time. If it is used in a trigger, it cannot be deleted.

In planning to delete an exception, all references to it should first be removed from the code of stored procedures, to avoid its absence causing errors.

Any user connected to the database can delete an exception.

Examples
1. Deleting exception ERROR_REFIN_RATE:

   DROP EXCEPTION ERROR_REFIN_RATE;

2. Deleting exception E_LARGE_VALUE:

   DROP EXCEPTION E_LARGE_VALUE;
5.12.5. RECREATE EXCEPTION

Used for
Creating a new custom exception or recreating an existing one

Available in
DSQL

Syntax

```
RECREATE EXCEPTION exception_name 'message'
```

Table 55. RECREATE 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>Error message. The maximum length is limited to 1,021 characters</td>
</tr>
</tbody>
</table>

The statement `RECREATE EXCEPTION` creates a new exception for use in PSQL modules. If an exception of the same name exists already, the `RECREATE EXCEPTION` statement will try to delete 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.

Any user connected to the database can [re]create an exception.

Example

Recreating the `E_LARGE_VALUE` exception:

```
RECREATE EXCEPTION E_LARGE_VALUE 'The value exceeds its limit';
```

See also

CREATE EXCEPTION, DROP EXCEPTION, CREATE OR ALTER EXCEPTION

5.13. COLLATION

5.13.1. CREATE COLLATION

Used for
Making a new collation for a supported character set available to the database

Available in
DSQL
Syntax

```
CREATE COLLATION collname
  FOR charset
     [FROM basecoll | FROM EXTERNAL ('extname')]
     [NO PAD | PAD SPACE]
     [CASE [IN]SENSITIVE]
     [ACCENT [IN]SENSITIVE]
     ['<specific-attributes>']

<specific-attributes> ::= <attribute> ; <attribute> ...

<attribute> ::= attrname=attrvalue
```

Table 56. 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 intl 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

If no FROM clause is present, Firebird will scan the .conf file(s) in the intl subdirectory for a collation with the name specified as the object of 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.

Specific Attributes

The available specific attributes are listed in the table below. Not all specific attributes apply to every collation, even if specifying them does not cause an error.

⚠️ Specific attributes 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). “UNI” stands for “UNICODE collations”.

---

Chapter 5. Data Definition (DDL) Statements

182
Table 57. 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>ICU-VERSION</td>
<td>default or M.m</td>
<td>UNI</td>
<td>Specifies the ICU library version to use. Valid values are the ones defined in the applicable &lt;intl_module&gt; element in intl/fbintl.conf. 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.

Any user connected to the database can use `CREATE COLLATION` to add a new collation.

Examples using `CREATE COLLATION`

1. Creating a collation using the name found in the `fbintl.conf` file (case-sensitive).
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

5.13.2. DROP COLLATION

Used for
Removing a collation from the database
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.

Any user connected to the database can use `DROP COLLATION` to remove a collation.

**Example using DROP COLLATION**
Deleting the `ES_ES_NOPAD_CI` collation.

```
DROP COLLATION ES_ES_NOPAD_CI;
```

**See also**
CREATE COLLATION

### 5.14. CHARACTER SET

#### 5.14.1. ALTER CHARACTER SET

**Used for**

Setting the default collation for a character set

**Available in**

DSQL

**Syntax**

```
ALTER CHARACTER SET charset
   SET DEFAULT COLLATION collation
```

**Table 59. ALTER CHARACTER SET Statement Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charset</td>
<td>The name of the character set</td>
</tr>
<tr>
<td>collation</td>
<td>The name of the collation</td>
</tr>
</tbody>
</table>
### Parameter Description

<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 statement `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.

#### Example of use

Setting the default `UNICODE_CI_AI` collation for the `UTF8` encoding.

```
ALTER CHARACTER SET UTF8
    SET DEFAULT COLLATION UNICODE_CI_AI;
```

### 5.15. ROLE

A role is a database object that packages a set of SQL 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 his login credentials in order to exercise the associated privileges. Any other privileges granted to the user are not affected by his 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.

#### 5.15.1. CREATE ROLE

**Used for**
Creating a new ROLE object

**Available in**
DSQL, ESQL
Syntax

```
CREATE ROLE rolename
```

**Table 60. CREATE ROLE Statement Parameter**

<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 user names as well. The system will not prevent the creation of a role whose name clashes with an existing user name but, if it happens, the user will be unable to connect to the database.

Any user connected to the database can create a role. The user that creates a role becomes its owner.

**Example**

Creating a role named `SELLERS`:

```
CREATE ROLE SELLERS;
```

**See also**

`DROP ROLE`, `GRANT`, `REVOKE`

### 5.15.2. ALTER ROLE

`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 procedure 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` in the `Security` chapter.

### 5.15.3. 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.

A role can be deleted by its owner or by an administrator.

Example

Deleting the role SELLERS:

```
DROP ROLE SELLERS;
```

See also

CREATE ROLE, GRANT, REVOKE

5.16. COMMENTS

Database objects and a database itself may contain 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.

5.16.1. COMMENT ON

Used for

Documenting metadata

Available in

DSQL


Syntax

```
COMMENT ON <object> IS {'sometext' | NULL}
```

```
<object> ::= DATABASE | <basic-type> objectname | COLUMN relationname.fieldname | PARAMETER procname.paramname
```

```
<basic-type> ::= CHARACTER SET | COLLATION | DOMAIN | EXCEPTION | EXTERNAL FUNCTION | FILTER | GENERATOR | INDEX | PROCEDURE | ROLE | SEQUENCE | TABLE | TRIGGER | VIEW
```

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Description</strong></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>procname</td>
<td>Name of stored procedure</td>
</tr>
<tr>
<td>paramname</td>
<td>Name of a stored procedure parameter</td>
</tr>
</tbody>
</table>

The **COMMENT ON** statement adds comments for database objects (metadata). Comments are saved to text fields of the **BLOB** type in the **RDB$DESCRIPTION** column of the corresponding system tables. Client applications can view comments from these fields.

- If you add an empty comment ("''"), it will be saved as **NULL** in the database.

The table or procedure owner and **Administrators** have the authority to use **COMMENT ON**.

**Examples using COMMENT ON**

1. Adding a comment for the current database
2. Adding a comment for the METALS table

```
COMMENT ON TABLE METALS IS 'Metal directory';
```

3. Adding a comment for the ISALLOY field in the METALS table

```
COMMENT ON COLUMN METALS.ISALLOY IS '0 = fine metal, 1 = alloy';
```

4. Adding a comment for a parameter

```
COMMENT ON PARAMETER ADD_EMP_PROJ.EMP_NO IS 'Employee ID';
```
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>]]
[FOR UPDATE [OF <columns>]]
[WITH LOCK]
[INTO <variables>]

<variables> ::= [:]varname [, [:]varname ...]
```

**Description**

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.
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:

```sql
select id, name, address
from contacts
```

Or, to retrieve all the columns:

```sql
select * from sales
```

In practice, the rows retrieved are often limited by a WHERE clause. The result set may be sorted by an ORDER BY clause, and FIRST, SKIP or ROWS may further limit the number of output rows. 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>`
- `<query-parameter>`
- `(<integer-expression>)`

*Table 62. Arguments for the FIRST and SKIP Clauses*
FIRST and SKIP are non-standard syntax
FIRST and SKIP are Firebird-specific, non-SQL-compliant keywords. You are advised to use the ROWS syntax wherever possible.

Description
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.

An error occurs when you use FIRST in subqueries. This query

```
DELETE FROM MYTABLE
WHERE ID IN (SELECT FIRST 10 ID FROM MYTABLE)
```

will delete all records from the table. The subquery retrieves 10 rows each time, deletes them and the operation is repeated until the table is empty. Keep it in mind! Or, better, use the ROWS clause in the DELETE statement.

Examples of FIRST/SKIP

The following query will return the first 10 names from the People table:
The columns list contains one or more comma-separated value expressions. Each expression provides a value for one output column. Alternatively, * ("select star") can be used to stand for all the columns in a relation (i.e. a table, view or selectable stored procedure).
### Syntax

```sql
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 63. 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>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 or aggregate function call 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>

### Description
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.

Aliases obfuscate 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:

```sql
select cust_id, cust_name, phone
from customers
where city = 'London'
```

A query featuring a concatenation expression and a function call in the columns list:

```sql
select 'Mr./Mrs.' || lastname, street, zip, upper(city)
from contacts
where date_last_purchase(id) = current_date
```

A query with two subselects:

```sql
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:

```sql
select p.fullname,
c.name as class,
m.name as mentor
join classes c on c.id = p.class
from pupils p
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:

```sql
select case upper(sex)
  when 'F' then 'Mrs.'
  when 'M' then 'Mr.'
  else ''
end as title,
lastname,
address
from employees
```

Querying a selectable stored procedure:

```sql
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:

```sql
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`):

```sql
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 any table or view, but can be derived from the expressions in the output columns alone. Another example is:

```sql
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:

```sql
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, 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. But 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 concentrates on single-source selects. *Joins* are discussed in a following section.
Syntax

SELECT ...
FROM <source>
[<joins>]
[...]

<source> ::= 
{ table
  | view
  | selectable-stored-procedure [(<args>)]
  | <derived-table>
  | <common-table-expression>
} [[AS] alias]

<derived-table> ::= 
( <select-statement> ) [[AS] alias] [(<column-aliases>)]

<common-table-expression> ::= 
WITH [RECURSIVE] <cte-def> [, <cte-def> ...]
<select-statement>

<cte-def> ::= name [(<column-aliases>)] AS (<select-statement>)

<column-aliases> ::= column-alias [, column-alias ...]

Table 64. 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</td>
</tr>
<tr>
<td>view</td>
<td>Name of a view</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 arguments</td>
</tr>
<tr>
<td>derived table</td>
<td>Derived table query expression</td>
</tr>
<tr>
<td>cte-def</td>
<td>Common table expression (CTE) definition, including an “ad hoc” name</td>
</tr>
<tr>
<td>select-statement</td>
<td>Any SELECT statement</td>
</tr>
<tr>
<td>column-aliases</td>
<td>Alias for a column in a relation, CTE or derived table</td>
</tr>
<tr>
<td>name</td>
<td>The “ad hoc” name for a CTE</td>
</tr>
<tr>
<td>alias</td>
<td>The alias of a data source (table, view, procedure, CTE, derived table)</td>
</tr>
</tbody>
</table>

Selecting FROM a table or view

When selecting from a single table or view, the FROM clause need not contain anything 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**

```
select id, name, sex, age from actors
where state = 'Ohio'
```

```
select * from birds
where type = 'flightless'
order by family, genus, species
```

```
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:

```
SELECT PEAR
FROM FRUIT;
```

```
SELECT FRUIT.PEAR
FROM FRUIT;
```

```
SELECT PEAR
FROM FRUIT F;
```

```
SELECT F.PEAR
FROM FRUIT F;
```

Incorrect use:

```
SELECT FRUIT.PEAR
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:

```sql
going 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:

```sql
going 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**

```sql
(<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.

**Sample 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. A “drill-down” query on the derived table returns the counts of fields and the counts of tables having each field count:
SELECT
  FIELD_COUNT,
  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
  (DES C R, 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 \texttt{COEFFS} which contains the coefficients of a number of quadratic equations we have to solve. It has been defined like this:

```sql
create table coeffs (  
a double precision not null,  
b double precision not null,  
c double precision not null,  
constraint chk_a_not_zero check (a <> 0)  
)
```

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 \texttt{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.

**Selecting FROM a CTE**

A common table expression or \textit{CTE} is a more complex variant of the derived table, but it is also more powerful. A preamble, starting with the keyword \texttt{WITH}, defines one or more named \textit{CTE}’s, each with an optional column aliases list. The main query, which follows the preamble, can then access these \textit{CTE}’s as if they were regular tables or views. The \textit{CTE}’s go out of scope once the main query has run to completion.

For a full discussion of \textit{CTE}’s, please refer to the section \textit{Common Table Expressions (WITH \ldots AS \ldots SELECT)}. 

---
The following is a rewrite of our derived table example as a CTE:

```sql
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. But we can now also eliminate the double calculation of sqrt(D) for every row:

```sql
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 SQRT function or passing the values of b, D and denom through an extra CTE). Incidentally, we could have done the same with derived tables, but that would involve nesting.

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 
  | view 
  | selectable-stored-procedure [(<args>)] 
  | <derived-table> 
  | <common-table-expression> 
} [[AS] alias]
```

```
<joins> ::= <join> [<join> ...]
```

```
<join> ::= 
[<join-type>] JOIN <source> <join-condition> 
| NATURAL [<join-type>] JOIN <source> 
| {CROSS JOIN | ,} <source>
```

```
<join-type> ::= INNER | {LEFT | RIGHT | FULL} [OUTER]
```

```
<join-condition> ::= ON <condition> | USING (<column-list>)
```

Table 65. Arguments for JOIN Clauses

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>Name of a table</td>
</tr>
<tr>
<td>view</td>
<td>name of a view</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>Reference, by name, to a derived table</td>
</tr>
<tr>
<td>common-table-expression</td>
<td>Reference, by name, to a common table expression (CTE)</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**

<table>
<thead>
<tr>
<th>ID</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>Just some text</td>
</tr>
<tr>
<td>235</td>
<td>Silence</td>
</tr>
</tbody>
</table>

**Table B**

<table>
<thead>
<tr>
<th>COD E</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>-23</td>
<td>56.7735</td>
</tr>
<tr>
<td>87</td>
<td>416.0</td>
</tr>
</tbody>
</table>

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>COD E</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>Just some text</td>
<td>87</td>
<td>416.0</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, this is rarely done.

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>COD E</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>ID</td>
<td>S</td>
<td>CODE</td>
<td>X</td>
</tr>
<tr>
<td>----</td>
<td>------------------------</td>
<td>------</td>
<td>---------</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;
```
Qualified joins

Qualified joins specify conditions for the combining of rows. This happens either explicitly in an ON clause or implicitly in a USING clause.

Syntax

\[
<\text{qualified-join}> ::= [<\text{join-type}>] \text{ JOIN } <\text{source}> <\text{join-condition}>
\]

\[
<\text{join-type}> ::= \text{INNER} \mid \{\text{LEFT} \mid \text{RIGHT} \mid \text{FULL}\} [\text{OUTER}]
\]

\[
<\text{join-condition}> ::= \text{ON } <\text{condition}> \mid \text{USING (} <\text{column-list}> <\text{)}>
\]

Explicit-condition joins

Most qualified joins have an 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 “=” operator. Joins like these are called 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:

```
select * from flotsam f
join jetsam j
on f.sea = j.sea
and f.ship = j.ship;
```

we can also write:

```
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:

```
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 a
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 that 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.

**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:

```sql
create table TA (  
a bigint,  
s varchar(12),  
is_date date  
);
```

```sql
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:

```sql
select * from TA
  natural join TB;
```

```sql
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.

Caution: 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.

**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;
```

**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.
<cross-join> ::= {CROSS JOIN | ,} <source>

Please notice that the comma syntax is deprecated! It is only supported to keep legacy code working and may disappear in some future version.

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
join TB on 1 = 1;
```

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:

```
select m.name, s.size, c.name
from materials m
cross join sizes s
cross join colors c;
```

**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:

```
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
Joins with stored procedures

If a join is performed with a stored procedure that is not correlated with other data streams via input parameters, there are no oddities. If correlation is involved, an unpleasant quirk reveals itself. The problem is that the optimizer denies itself any way to determine the interrelationships of the input parameters of the procedure from the fields in the other streams:

```sql
SELECT *
FROM MY_TAB
JOIN MY_PROC(MY_TAB.F) ON 1 = 1;
```

Here, the procedure will be executed before a single record has been retrieved from the table, MY_TAB. The `isc_no_cur_rec` error (no current record for fetch operation) is raised, interrupting the execution.

The solution is to use syntax that specifies the join order explicitly:

```sql
SELECT *
FROM MY_TAB
LEFT JOIN MY_PROC(MY_TAB.F) ON 1 = 1;
```

This forces the table to be read before the procedure and everything works correctly.

This quirk has been recognised as a bug in the optimizer and will be fixed in the next version of Firebird.

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>  
  [...]  
```

```
<search-condition> ::=  
  a boolean expression returning  
  TRUE, FALSE or possibly UNKNOWN (NULL)
```

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;
```

```
select * from persons  
  where birthyear in (1880, 1881)  
    or birthyear between 1891 and 1898;
```

```
select name, street, borough, phone  
  from schools s  
  where exists (select * from pupils p where p.school = s.id)  
  order by borough, street;
```

```
select * from employees  
  where salary >= 10000 and position <> 'Manager';
```

```
select name from wrestlers  
  where region = 'Europe'  
    and weight > all (select weight from shot_putters  
    where region = 'Africa');
```
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>
<tr>
<td>Gerry</td>
<td>21</td>
</tr>
<tr>
<td>Hadassah</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>Isaac</td>
<td>6</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
```

you will get the names Bob E., Eve and Gerry. These children all have not more than 10 marbles.
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
where city = ? and class = ?;
```

```sql
select * from pants
where model = :model and size = :size and color = :col;
```

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

```
SELECT ... FROM ...
  GROUP BY <grouping-item> [, <grouping-item> ...]
  [HAVING <grouped-row-condition>]
...
```

```
<grouping-item> ::= 
  <non-aggr-select-item> 
  | <non-aggr-expression>
<non-aggr-select-item> ::= 
  column-copy
  | column-alias
  | column-position
```

Table 66. 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>
<tr>
<td>column-position</td>
<td>The position number, in 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. But 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;
```

and 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:

```
select class, 
    count(*) as num_boys, 
    avg(age) as boys_avg_age 
from students 
where sex = 'M' 
group by class;
```

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:

```
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>
</tbody>
</table>
```
Each row in the result set corresponds to one particular combination of the variables 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 subclause 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 alternative.
- 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:
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:

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:

select class,
    count(*) as num_boys,
    avg(age) as boys_avg_age
from students
where sex = 'M'
group by class
having class starting with '3';

Better would be to move this condition to the WHERE clause:

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.
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.

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:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>Table name or its alias</td>
</tr>
<tr>
<td>view</td>
<td>View name</td>
</tr>
<tr>
<td>index</td>
<td>Index name</td>
</tr>
</tbody>
</table>
If there's a WHERE or a HAVING clause, you can specify the index to be used for finding matches:

```sql
select * from students
where class = '3C'
plan (students index (ix_stud_class));
```

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:

```sql
select * from students
where class >= '3'
plan (students order pk_students index (ix_stud_class))
order by id;
```

ORDER and INDEX can be combined:

```sql
select * from students
where class >= '3'
plan (students order pk_students index (ix_stud_class))
order by id;
```

It is perfectly OK if ORDER and INDEX specify the same index:

```sql
select * from students
where class >= '3'
plan (students order ix_stud_class index (ix_stud_class))
order by class;
```

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;
```

Or when an index is used for the search:
select * from students
    where class >= '3'
    plan sort (students index (ix_stud_class))
    order by name;

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:

select * from freshmen
    plan (freshmen students natural);

Or, for instance:

select * from freshmen
    where id > 10
    plan sort (freshmen students index (pk_students))
    order by name desc;

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));

The same join, sorted on an indexed column:

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;

And on a non-indexed column:
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;

With a search added:

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;

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;

If there is no index available to match the join criteria (or if you don’t want to use it), the plan must 
first sort both streams on their join column(s) and then merge them. This is achieved with the SORT 
directive (which we’ve already met) 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:
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.

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

A UNION 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

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:
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
select r.*, 'LP'
from records r
union
select c.*, 'MC'
from cassettes c
order by 3, 2  -- artist, title;
```

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 DISTINCT. With ALL, these people would appear twice.

```sql
select name, phone from translators
union distinct
select name, telephone from proofreaders;
```

A UNION within a subquery:

```sql
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;
```
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 INSERT statements. To specify a sorting order for the set specification, an ORDER BY clause is used.

Syntax

```sql
SELECT ... FROM ...
...
ORDER BY <ordering-item> [, <ordering-item> ...]
```

```sql
<ordering-item> ::= 
{col-name | col-alias | col-position | <expression>
[COLLATE collation-name]
[ASC[ENDING] | DESC[ENDING]]
[NULLS {FIRST|LAST}]
```

<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>

Description

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 there. 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 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 the 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 one that has been set explicitly in the column’s definition.

NULLs Position

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$CHARACTER_SET_ID**, **RDB$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,
    RDB$COLLATION_NAME AS NAME
FROM RDB$COLLATIONS
ORDER BY 1, 2;
```

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:

```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.
```
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**

DSQL, PSQL

**Syntax**

```
SELECT <columns> FROM ...
WHERE ...
[ORDER BY ...]
ROWS m [TO n]
```

<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>

**Description**

Limits the amount of rows returned by the `SELECT` statement to a specified number or range.

The `FIRST` and `SKIP` clauses do the same job as `ROWS`, but neither are SQL-compliant. Unlike `FIRST` and `SKIP`, 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.

- Both `FIRST/Skip` and `ROWS` can 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 random 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` >= `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.

Mixing `ROWS` and FIRST/SKIP

`ROWS` syntax cannot be mixed with FIRST/SKIP syntax 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 ROWS**

The following examples rewrite the examples used in the section about FIRST and SKIP, 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
ROWS 1 TO 10;
```

or its equivalent

```
SELECT id, name
FROM People
ORDER BY name ASC
ROWS 10;
```

Return all records from the PEOPLE table except for the first 10 names:

```
SELECT id, name
FROM People
ORDER BY name ASC
ROWS 11 TO (SELECT COUNT(*) FROM People);
```

And this query will return the last 10 records (pay attention to the parentheses):

```
SELECT id, name
FROM People
ORDER BY name ASC
ROWS (SELECT COUNT(*) - 9 FROM People)
TO (SELECT COUNT(*) FROM People);
```

This one will return rows 81-100 from the PEOPLE table:

```
SELECT id, name
FROM People
ORDER BY name ASC
ROWS 81 TO 100;
```

ROWS can also be used with the **UPDATE** and **DELETE** statements.
6.1.11. FOR UPDATE [OF]

Syntax

```
SELECT ... FROM single_table
  [WHERE ...]
  [FOR UPDATE [OF ...]]
```

FOR UPDATE does not do what it suggests. Its 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 OF sub-clause does not do anything at all.

6.1.12. WITH LOCK

Available in

DSQL, PSQL

Used for

Limited pessimistic locking

Description:

WITH LOCK provides a limited explicit pessimistic locking capability for cautious use in conditions where the affected row set is:

a. extremely small (ideally, a 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.

Syntax

```
SELECT ... FROM single_table
  [WHERE ...]
  [FOR UPDATE [OF ...]]
  WITH LOCK
```

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 70. How **TPB** settings affect explicit locking*

<table>
<thead>
<tr>
<th><strong>TPB mode</strong></th>
<th><strong>Behaviour</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>isc_tpb_consistency</strong></td>
<td>Explicit locks are overridden by implicit or explicit table-level locks and are ignored.</td>
</tr>
<tr>
<td><strong>isc_tpb_concurrency + isc_tpb_nowait</strong></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><strong>isc_tpb_concurrency + isc_tpb_wait</strong></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></td>
<td>If an active transaction is holding ownership on this record (via explicit locking or by a normal optimistic write-lock) the transaction attempting the explicit lock waits for the outcome of the blocking transaction and, when it finishes, attempts to get the lock on the record again. This means that, if the blocking transaction committed a modified version of this record, an update conflict exception will be raised.</td>
</tr>
<tr>
<td><strong>isc_tpb_read_committed + isc_tpb_nowait</strong></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>
</tbody>
</table>
**TPB mode**

<table>
<thead>
<tr>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>isc_tpb_readcommitted + isc_tpb_wait                                                                                       If there is an active transaction holding ownership on this record (via explicit locking or by a 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-names>`

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, where 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:

```
SELECT * FROM DOCUMENT WHERE ID=? WITH LOCK;
```

ii. Multiple rows, one-by-one processing with DSQL cursor:

```
SELECT * FROM DOCUMENT WHERE PARENT_ID=? FOR UPDATE WITH LOCK;
```

6.1.13. **INTO**

*Used for*

Passing SELECT output into variables

*Available in*

PSQL

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. 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.

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.

Examples

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
    into min_amt, avg_amt, max_amt;
```

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.

A PSQL trigger that retrieves two values as a BLOB field (using the LIST() function) and assigns it INTO a third field:

```
select list(name, ',', '')
    from persons p
    where p.id in (new.father, new.mother)
    into new.parentnames;
```
6.1.14. Common Table Expressions ("WITH … AS … SELECT")

Available in
DSQL, PSQL

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.

Syntax

<table>
<thead>
<tr>
<th>cte-construct</th>
<th>cte-defs</th>
<th>main-query</th>
</tr>
</thead>
<tbody>
<tr>
<td>cte-defs ::= WITH [RECURSIVE] cte [, cte] ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cte ::= name [(&lt;column-list&gt;)] AS (&lt;cte-stmt&gt;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;column-list&gt; ::= column-alias [, column-alias]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 71. 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>
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 (  
  select * from proj_dept_budget b  
  where d.dept_no = b.dept_no
);
```

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
```
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’.

**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

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,  
        D.DEPARTMENT,  
        H.INDENT || ' '  
    FROM DEPARTMENT D  
    JOIN DEPT_TREE H ON H.HEAD_DEPT = D.DEPT_NO  
)  
SELECT  
    D.DEPT_NO,  
    D.INDENT || D.DEPARTMENT DEPARTMENT,  
    DYB_2008.BUDGET AS BUDGET_08,  
    DYB_2009.BUDGET AS BUDGET_09  
FROM DEPT_TREE 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);

The next example returns the pedigree of a horse. The main difference is that recursion occurs simultaneously in two branches of the pedigree.

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

\[
\text{INSERT INTO } \text{target} \\
\{\text{DEFAULT VALUES | [\{(<column_list>)] <value_source>\}} \\
[\text{RETURNING <returning_list> [INTO <variables>]]}
\]

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

<value_source> ::= VALUES (<value_list>) | <select_stmt>

<value_list> ::= <value> [, <value> ...]

<returning_list> ::= <ret_value> [, <ret_value> ...]

<ret_value> ::= colname | <value>

<variables> ::= [:]varname [, [:]varname ...]

Table 72. Arguments for the 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 to which a new row, or batch of rows, should be added</td>
</tr>
<tr>
<td>colname</td>
<td>Column in the table or view</td>
</tr>
<tr>
<td>value</td>
<td>An expression whose value is used for inserting into the table or for returning</td>
</tr>
<tr>
<td>ret_value</td>
<td>The expression to be returned in the RETURNING clause</td>
</tr>
<tr>
<td>varname</td>
<td>Name of a PSQL local variable</td>
</tr>
</tbody>
</table>

Description

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**

```sql
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**

```sql
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.

**The “Unstable Cursor” Problem**

In Firebird, up to and including Firebird 2.5, it is necessary to be aware of an implementation fault that affects this style of inserts when the objective is to duplicate rows in the same table. For example,

```sql
INSERT INTO T
    SELECT * FROM T;
```

known affectionately as the “infinite insertion loop”, will continuously select rows and insert them, over and over, until the system runs out of storage space.

This is a quirk that affects all data-changing DML operations, with a variety of effects. It happens because, in the execution layers, DML statements use implicit cursors for performing the
operations. Thus, using our simple example, execution works as follows:

```
FOR SELECT <values> FROM T INTO <tmp_vars>
    DO
    INSERT INTO T VALUES (<tmp_vars>);
```

The implementation results in behaviour that does not accord with the SQL standards. Future versions of Firebird will comply with the standard.

### 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.

**ALERT: 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

```
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 only supported for 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 text string of no more than 32767 bytes.

   If the value is not a string literal, beware of concatenations, as the output from the expression may exceed the maximum length.

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
**Syntax**

```
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>]]
```

<returning_list> ::= <ret_value> [, <ret_value> ...]

<ret_value> ::=
  colname | NEW.colname
  | OLD.colname
  | <value>

<variables> ::= [:]varname [, [:]varname ...]

**Table 73. Arguments for the UPDATE 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 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 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 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>
<tr>
<td>varname</td>
<td>Name of a PSQL local variable</td>
</tr>
</tbody>
</table>

**Description**

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.

Examples

Correct usage:

```sql
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:

```sql
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:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

The statement:

```sql
UPDATE tset SET a = 5, b = a;
```

will change the values to:
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).

It was not always like that. Before version 2.5, columns got their new values immediately upon assignment. It was non-standard behaviour that was fixed in version 2.5.

To maintain compatibility with legacy code, the configuration file firebird.conf includes the parameter OldSetClauseSemantics, that can be set True (1) to restore the old, bad behaviour. It is a temporary measure—the parameter will be removed in the future.

### 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**

```sql
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;
```

**The “Unstable Cursor” Problem**

In Firebird, up to and including Firebird 2.5, it is necessary to be aware of an implementation fault
that affects updates when the WHERE conditions use the IN `(select-expr)` and the `select-expr` is of
the form `SELECT FIRST n or SELECT … ROWS`. For example:

```sql
UPDATE T
SET ...
WHERE ID IN (SELECT FIRST 1 ID FROM T);
```

known affectionately as the “infinite update loop”, will continuously update rows, over and over,
and give the impression that the server has hung.

Quirks like this can affect any data-changing DML operations, most often when the selection
conditions involve a subquery. Cases have been reported where sort order interferes with
expectations, without involving a subquery. It happens because, in the execution layers, instead of
establishing a stable “target set” and then executing the data changes to each set member, DML
statements use implicit cursors for performing the operations on whatever row currently meets the
conditions, without knowledge of whether that row formerly failed the condition or was updated
already. Thus, using a simple example pattern:

```sql
UPDATE T SET <fields> = <values>
WHERE <conditions>
```

the execution works as:

```sql
FOR SELECT <values> FROM T
WHERE <conditions>
INTO <tmp_vars> AS CURSOR <cursor>
DO
  UPDATE T SET <fields> = <tmp_vars>
  WHERE CURRENT OF <cursor>
```

Firebird's implementation does not accord with the SQL standards, which require that a stable set
be established before any data are changed. Firebird 3 and higher will comply with the standard.
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. 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.

When a value is returned and assigned to a NEW context variable, it is not valid to use a colon prefix on it. For example, this is invalid:

```
... into :var1, :var2, :new.id
```

and this is valid:

```
... into :var1, :var2, new.id
```

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 text string of at most 32767 bytes. Please notice: if the value is not a string literal, beware of concatenations, as these may exceed the maximum length.
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 <values> [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 74. 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>

Description

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 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 is raised. This behaviour may change in a later version of Firebird.

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.

```
UPDATE OR INSERT INTO Cows (Name, Number, Location)
VALUES ('Suzy Creamcheese', 3278823, 'Green Pastures')
MATCHING (Number)
RETURNING rec_id into :id;
```

The “Unstable Cursor” Problem

Because of the way the execution of data-changing DML is implemented in Firebird, up to and including Firebird 2.5, the sets targeted for updating sometimes produce unexpected results. For more information, refer to The Unstable Cursor Problem in the UPDATE section.

6.5. DELETE

*Used for*
Deleting rows from a table or view

*Available in*
DSQL, ESQL, PSQL
**Syntax**

```
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>]]

<returning_list> ::= <ret_value> [, <ret_value> ...]
<ret_value> ::= colname | <value>
<variables> ::= [:]varname [, [:]varname ...]
```

**Table 75. 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 RETURNING 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>

**Description**

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:
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:
delete from Cities C where Cities.name starting 'Alex';

**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 \( <= \) 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:

```sql
DELETE FROM Purchases
ORDER BY date ROWS 1;
```

Deleting the highest custno(s):

```sql
DELETE FROM Sales
ORDER BY custno DESC ROWS 1 to 10;
```

Deleting all sales, ORDER BY clause pointless:

```sql
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:

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.

Notes

- In DSQL, a statement with RETURNING always returns a singleton, never a multi-row set. 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

DELETE FROM Scholars
WHERE firstname = 'Henry' and lastname = 'Higgins'
RETURNING lastname, fullname, id;

DELETE FROM Dumbbells
ORDER BY iq DESC
ROWS 1
RETURNING lastname, iq into :lname, :iq;

The “Unstable Cursor” Problem

Because of the way the execution of data-changing DML is implemented in Firebird, up to and including this version, the sets targeted for deletion sometimes produce unexpected results. For more information, refer to The Unstable Cursor Problem in the UPDATE section.
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>
    [ WHEN MATCHED
      THEN UPDATE SET colname = <value> [, colname = <value> ...]]
    [ WHEN NOT MATCHED
      THEN INSERT [(<columns>)] VALUES (<values>)]
```

<table>
<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>tablename</td>
<td>Table or view name</td>
</tr>
<tr>
<td>select-stmt</td>
<td>Select statement of the derived table</td>
</tr>
<tr>
<td>colname</td>
<td>Name of a column in the target relation</td>
</tr>
<tr>
<td>value</td>
<td>The value assigned to a column in the target table. It is an expression that may be a literal value, a PSQl variable, a column from the source or a compatible context variable</td>
</tr>
</tbody>
</table>
</table>

Description

The MERGE statement merges data into a 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, or neither.
The action taken depends on the supplied join condition and the **WHEN** clause(s). The condition will typically contain a comparison of fields in the source and target relations.

### Notes

At least one **WHEN** clause must be present. Only one of each **WHEN** clause can be supplied. This will change in the next major version of Firebird, when compound matching conditions will be supported.

**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 that 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).

### Examples

```
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);
```

```
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);
```
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);

The “Unstable Cursor” Problem
Because of the way the execution of data-changing DML is implemented in Firebird, up to and including this version, the sets targeted for merging sometimes produce unexpected results. For more information, refer to The Unstable Cursor Problem in the UPDATE section.

6.7. EXECUTE PROCEDURE

Used for
Executing a stored procedure

Available in
DSQL, ESQL, PSQL

Syntax

EXECUTE PROCEDURE procname
    [<inparam> [, <inparam> ...]] | [(<inparam> [, <inparam> ...])]    
    [RETURNING VALUES <outvar> [, <outvar> ...] | (<outvar> [, <outvar> ...])]    

<outvar> ::= [:]varname

Table 77. Arguments for the EXECUTE PROCEDURE Statement Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>proiname</td>
<td>Name of the stored procedure</td>
</tr>
<tr>
<td>inparam</td>
<td>An expression evaluating to the declared data type of an input parameter</td>
</tr>
<tr>
<td>varname</td>
<td>A PSQL variable to receive the return value</td>
</tr>
</tbody>
</table>

Description

Executes an 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.
6.7.1. “Executable” Stored Procedure

The 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 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 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 EXECUTE PROCEDURE in DSQL.

Invoking the other style of stored procedure—a “selectable” one—is possible with 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 SELECT statement, producing output that behaves like a virtual table.

Notes

- 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 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 RETURNING_VALUES may be optionally enclosed in parentheses and their use is recommended.

- When DSQL applications call EXECUTE PROCEDURE using the Firebird API or some form of wrapper for it, a buffer is prepared to receive the output row and the RETURNING_VALUES clause is not used.

6.7.2. Examples of EXECUTE PROCEDURE

In PSQL, with optional colons and without optional parentheses:

EXECUTE PROCEDURE MakeFullName
  :FirstName, :MiddleName, :LastName
RETURNING_VALUES :FullName;

In Firebird’s command-line utility isql, with literal parameters and optional parentheses:

EXECUTE PROCEDURE MakeFullName ('J', 'Edgar', 'Hoover');

In isql, RETURNING_VALUES is not used. Any output values are captured by the application and displayed automatically.
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))]  
  AS  
  [declare_item]  
  BEGIN  
  [PSQL statements]  
  END
```

- `<inparams>` ::=  `<param_decl> = ? [, `<inparams>` ]`
- `<outparams>` ::=  `<param_decl> [, `<outparams>` ]`
- `<param_decl>` ::=  `paramname <type> [NOT NULL] [COLLATE collation]`
- `<type>` ::=  `<datatype> | [TYPE OF] domain | TYPE OF COLUMN rel.col`

- `<datatype>` ::=  
  - `{SMALLINT | INTEGER | BIGINT}`
  - `{FLOAT | DOUBLE PRECISION}`
  - `{DATE | TIME | TIMESTAMP}`
  - `{DECIMAL | NUMERIC} [(precision [, scale])]
  - `{CHAR | CHARACTER} [VARYING] | VARCHAR} [(size)]`
  - `{CHARACTER SET charset]`
  - `{NCHAR | NATIONAL {CHARACTER | CHAR}} [VARYING] [(size)]`
  - `BLOB [SUB_TYPE {subtype_num | subtype_name}]`
  - `[SEGMENT SIZE seglen] [CHARACTER SET charset]`
  - `BLOB [(seglen [, subtype_num])]

- `<declarations>` ::=  `<declare_item> [declare_item] ...
- `<declare_item>` ::=  `<declare_var>; | <declare_cursor>;

**Table 78. 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>declarations</td>
<td>A section for declaring local variables and named cursors</td>
</tr>
<tr>
<td>declare_var</td>
<td>Local variable declaration</td>
</tr>
<tr>
<td>declare_cursor</td>
<td>Declaration of a named cursor</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>datatype</td>
<td>SQL data type</td>
</tr>
<tr>
<td>collation</td>
<td>Collation sequence</td>
</tr>
<tr>
<td>domain</td>
<td>Domain</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</td>
</tr>
<tr>
<td>precision</td>
<td>Precision. From 1 to 18</td>
</tr>
<tr>
<td>scale</td>
<td>Scale. From 0 to 18. It must be less than or equal to precision</td>
</tr>
<tr>
<td>size</td>
<td>The maximum size of a string, in characters</td>
</tr>
<tr>
<td>charset</td>
<td>Character set</td>
</tr>
<tr>
<td>subtype_num</td>
<td>BLOB subtype number</td>
</tr>
<tr>
<td>subtype_name</td>
<td>BLOB subtype mnemonic name</td>
</tr>
<tr>
<td>seglen</td>
<td>Segment size, it cannot be greater than 65,535</td>
</tr>
</tbody>
</table>

**Description**

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.

**Examples**

This example injects the numbers 0 through 127 and their corresponding ASCII characters into the table **ASCII**.

```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
```

The next example calculates the geometric mean of two numbers and returns it to the user:
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. 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.

Again, it depends on the client software if and how you can set the parameter values.

### 6.8.1. 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.
6.8.2. 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 slight modifications to syntax in some cases.

7.1. Elements of PSQL

A procedural extension may contain declarations of local variables 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, trigger or block) use parameters, only named parameters can be used and they must “exist” before the statements can use them. They can be made available by being declared either as input or output parameters in the module’s header or as local variables, in DECLARE [VARIABLE] statements at the bottom of the 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. 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 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 parameters and variables that are used in the body. Stored procedures and PSQL blocks may have input and output parameters. 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 body of a PSQL module is a block of statements that run in a logical sequence, like a program. A block of statements is contained within a BEGIN and an END statement. The main BEGIN…END block may contain any number of other BEGIN…END blocks, both embedded and sequential. All statements except BEGIN and END are terminated by semicolons (‘;’). No other character is valid for use as a terminator for PSQL statements.
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 79. 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:
7.2. Stored Procedures

A stored procedure is a program stored in the database metadata for execution on the server. A stored procedure can be called by stored procedures (including itself), 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 statement.
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.

Syntax (partial)

```
CREATE PROCEDURE procname
  [(<inparam> [, <inparam> ...])]
  [RETURNS (<outparam> [, <outparam> ...])]
AS
  [<declarations>]
BEGIN
  [<PSQL_statements>]
END
```

The header of a stored procedure must contain the procedure name, and it must be unique among the names of stored procedures, tables, and views. It may also define some input and output parameters. Input parameters are listed after the procedure name inside a pair of brackets. Output parameters, which are mandatory for selectable procedures, are bracketed inside one RETURNS clause.

The final item in the header (or the first item in the body, depending on your opinion of where the border lies) is one or more declarations of any local variables and/or named cursors that your procedure might require.

Following the declarations is the main BEGIN…END block that delineates the procedure's PSQL code. Within that block could be PSQL and DML statements, flow-of-control blocks, sequences of other BEGIN…END blocks, including embedded blocks. Blocks, including the main block, may be empty and
the procedure will still compile. It is not unusual to develop a procedure in stages, from an outline.

For more information about creating stored procedures
See CREATE PROCEDURE in Chapter 5, Data Definition (DDL) Statements.

7.2.4. Modifying a Stored Procedure

An existing stored procedure can be altered, to change the sets of input and output parameters and anything in the procedure body.

Syntax (partial)

ALTER PROCEDURE procname
   [((<inparam> [, <inparam> ...])]
   [RETURNS (<outparam> [, <outparam> ...])]
   AS
   [<declarations>]
BEGIN
   [<PSQL_statements>]
END

For more information about modifying stored procedures
See ALTER PROCEDURE, CREATE OR ALTER PROCEDURE, RECREATE PROCEDURE, in Chapter 5, Data Definition (DDL) Statements.

7.2.5. Deleting a Stored Procedure

The DROP PROCEDURE statement is used to delete stored procedures.

Syntax (complete)

DROP PROCEDURE <em>procname</em>

For more information about deleting stored procedures
See DROP PROCEDURE in Chapter 5, Data Definition (DDL) Statements.

7.3. Stored Functions

Stored PSQL scalar functions are not supported in this version but they are coming in Firebird 3. In Firebird 2.5 and below, you can instead write a selectable stored procedure that returns a scalar result and SELECT it from your DML query or subquery.
Example

```sql
SELECT
    PSQL_FUNC(T.col1, T.col2) AS col3,
    col3
FROM T
```

can be replaced with:

```sql
SELECT
    (SELECT output_column FROM PSQL_PROC(T.col1)) AS col3,
    col2
FROM T
```
or

```sql
SELECT
    output_column AS col3,
    col2,
FROM T
LEFT JOIN PSQL_PROC(T.col1)
```

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 variable and cursor declarations; and a block of PSQL statements follows.

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> ...])]
AS
    [<declarations>]
BEGIN
    [<PSQL_statements>]
END
```

Table 80. PSQL Block Parameters
### 7.5. 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.

One trigger applies to exactly one table or view and only one phase in an event (BEFORE or AFTER the event). A single 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 some of them, a default transaction is started.

#### 7.5.1. 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.

#### 7.5.2. DML Triggers

DML triggers are those that fire when a DML operation changes the state of data: modifies rows in tables, inserts new rows or deletes 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.

**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

### 7.5.3. Database Triggers

A trigger associated with a database or transaction event can be defined for the following events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Trigger Type</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connecting to a database</td>
<td>ON CONNECT</td>
<td>Before the trigger is executed, a default transaction is automatically started</td>
</tr>
<tr>
<td>Disconnecting from a database</td>
<td>ON DISCONNECT</td>
<td>Before the trigger is executed, a default transaction is automatically started</td>
</tr>
<tr>
<td>When a transaction is started</td>
<td>ON TRANSACTION START</td>
<td>The trigger is executed in the current transaction context</td>
</tr>
<tr>
<td>When a transaction is committed</td>
<td>ON TRANSACTION COMMIT</td>
<td>The trigger is executed in the current transaction context</td>
</tr>
<tr>
<td>When a transaction is cancelled</td>
<td>ON TRANSACTION ROLLBACK</td>
<td>The trigger is executed in the current transaction context</td>
</tr>
</tbody>
</table>
7.5.4. Creating Triggers

Syntax

CREATE TRIGGER trigname {
  <relation_trigger_legacy>
  | <relation_trigger_sql2003>
  | <database_trigger> }
AS
  [<declarations>]
BEGIN
  [<PSQL_statements>]
END

<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]

<mutation_list> ::= 
  <mutation> [OR <mutation> [OR <mutation>]]

<mutation> ::= { INSERT | UPDATE | DELETE }

<db_event> ::= 
  CONNECT 
  | DISCONNECT 
  | TRANSACTION START 
  | TRANSACTION COMMIT 
  | TRANSACTION ROLLBACK

The header must contain a name for the trigger that is unique among trigger names. It must include the event or events that will fire the trigger. Also, for a DML trigger it is mandatory to specify the event phase and the name of the table or view that is to “own” the trigger.

The body of the trigger can be headed by the declarations of local variables and cursors, if any. Within the enclosing main BEGIN…END wrapper will be one or more blocks of PSQL statements, which may be empty.
For more information about creating triggers
See "CREATE TRIGGER" in Chapter 5, Data Definition (DDL) Statements.

### 7.5.5. Modifying Triggers

Altering the status, phase, table or view event(s), firing position and code in the body of a DML trigger are all possible. However, you cannot modify a DML trigger to convert it to a database trigger, nor vice versa. Any element not specified is left unchanged by ALTER TRIGGER. The alternative statements CREATE OR ALTER TRIGGER and RECREATE TRIGGER will replace the original trigger definition entirely.

**Syntax**

```sql
ALTER TRIGGER trigname
  [ACTIVE | INACTIVE]
  [{BEFORE | AFTER} <mutation_list> | ON <db_event>]
  [POSITION number]
  [AS]
  [BEGIN]
  [AS]
  BEGIN
  [AS]
  [BEGIN]
  [AS]
  END
```  

<mutation_list> ::=  

<mutation> ::= { INSERT | UPDATE | DELETE }

<db_event> ::=  

{ CONNECT  
| DISCONNECT  
| TRANSACTION START  
| TRANSACTION COMMIT  
| TRANSACTION ROLLBACK }

For more information about modifying triggers
See ALTER TRIGGER, CREATE OR ALTER TRIGGER, RECREATE TRIGGER in Chapter 5, Data Definition (DDL) Statements.

### 7.5.6. Deleting a Trigger

The DROP TRIGGER statement is used to delete triggers.

**Syntax (complete)**

```sql
DROP TRIGGER triname
```
7.6. 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 code and it should never be applied to context variables.

7.6.1. Assignment Statements

*Used for*

Assigning a value to a variable

*Available in*

PSQL

*Syntax*

\[ \text{varname} = \text{<value_expr>} \]

*Table 81. 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 equivalence 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 or to external functions (UDFs).
Example using assignment statements

```sql
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

7.6.2. DECLARE CURSOR

Used for
Declaring a named cursor

Available in
PSQL

Syntax

```
DECLARE [VARIABLE] cursorname CURSOR FOR (<select>) [FOR UPDATE]
```

Table 82. DECLARE CURSOR Statement Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursorname</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 walk 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.

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 checks it 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:

  ```sql
  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.

  **Attention!**

  If the value of a PSQL variable used in the SELECT statement changes during the loop, its new value may (but not always) be used for the remaining rows. It is better to avoid having such situations arise unintentionally. If you really need this behaviour, you should test your code carefully to be certain that you know exactly how changes in the variable affect the result.

  Note particularly that the behaviour may depend on the query plan, specifically on the indexes being used. No strict rules are in place for situations like this currently, but that could change in future versions of Firebird.

Examples Using Named Cursors

1. Declaring a named cursor in the trigger.
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. A collection of scripts for creating views with a PSQL block using named 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 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
    CLOSE CUR_R;
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
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
OPEN, FETCH, CLOSE

7.6.3. DECLARE VARIABLE

Used for
Declaring a local variable

Available in
PSQL
Syntax

```sql
DECLARE [VARIABLE] varname
{<datatype> | domain | TYPE OF {domain | COLUMN rel.col}
[NOT NULL] [CHARACTER SET charset] [COLLATE collation]
[DEFAULT | = ] <initvalue>];

<datatype> ::= 
{SMALLINT | INTEGER | BIGINT}
| {FLOAT | DOUBLE PRECISION}
| {DATE | TIME | TIMESTAMP}
| {DECIMAL | NUMERIC} [(precision [, scale])]
| {CHAR | CHARACTER [VARYING] | VARCHAR} [(size)]
[CHARACTER SET charset]
| {NCHAR | NATIONAL {CHARACTER | CHAR}} [VARYING]
[ (size)]
| BLOB [SUB_TYPE {subtype_num | subtype_name}]
[SEGMENT SIZE seglen] [CHARACTER SET charset]
| BLOB [(seglen [, subtype_num])]}

<initvalue> ::= <literal> | <context_var>
```

Table 83. 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>datatype</td>
<td>An SQL data type</td>
</tr>
<tr>
<td>domain</td>
<td>The name of an existing domain in this database</td>
</tr>
<tr>
<td>rel.col</td>
<td>Relation name (table or view) in this database and the name of a column in that relation</td>
</tr>
<tr>
<td>precision</td>
<td>Precision. From 1 to 18</td>
</tr>
<tr>
<td>scale</td>
<td>Scale. From 0 to 18, it must be less than or equal to precision</td>
</tr>
<tr>
<td>size</td>
<td>The maximum size of a string in characters</td>
</tr>
<tr>
<td>subtype_num</td>
<td>BLOB subtype number</td>
</tr>
<tr>
<td>subtype_name</td>
<td>BLOB subtype mnemonic name</td>
</tr>
<tr>
<td>seglen</td>
<td>Segment size, not greater than 65,535</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>
<tr>
<td>charset</td>
<td>Character set</td>
</tr>
<tr>
<td>collation</td>
<td>Collation sequence</td>
</tr>
</tbody>
</table>

Chapter 7. Procedural SQL (PSQL) Statements

287
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.

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 and 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

The variable can be constrained `NOT NULL` if required. If a domain has been specified as the data type and already carries the `NOT NULL` constraint, it will not be necessary. With the other forms, including use of a domain that is nullable, the `NOT NULL` attribute should 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.

Be sure to use this clause for any variables that are constrained to be `NOT NULL` and do not otherwise have a default value available.

Examples of various ways to declare local variables
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

7.6.4. BEGIN … END

Used for
Delineating a block of statements

Available in
PSQL

Syntax

\[
<\text{block}> ::= \\
\text{BEGIN} \\
  [<\text{compound\_statement}>] \\
  … \\
\text{END}
\]

\[
<\text{compound\_statement}> ::= \{<\text{block}> | <\text{statement}>;\}
\]

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 to unlimited depth. They may be empty, allowing them to act as stubs, without the need to write dummy statements.

The BEGIN and END statements have no line terminators. 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.

**Example**

A sample procedure from the `employee.fdb` database, showing simple usage of `BEGIN...END` blocks:


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
END^  
SET TERM ;^  

See also
EXIT, LEAVE, SET TERM
7.6.5. **IF … THEN … ELSE**

*Used for*

Conditional jumps

*Available in*

PSQL

*Syntax*

```sql
IF (<condition>)
    THEN <compound_statement>
   [ELSE <compound_statement>]
```

*Table 84. IF … THEN … ELSE Parameters*

<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>
<tr>
<td>single_statement</td>
<td>A single statement terminated with a semicolon</td>
</tr>
<tr>
<td>compound_statement</td>
<td>Two or more statements wrapped in BEGIN … END</td>
</tr>
</tbody>
</table>

The conditional jump statement **IF … THEN** is used to branch the execution process in a PSQL module. The condition is always enclosed in parentheses. If it 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.
PSQL does not provide multi-branch jumps, such as `CASE` or `SWITCH`. Nevertheless, the `CASE` search statement from DSQL is available in PSQL and is able to satisfy at least some use cases in the manner of a switch:

```
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**

```
...  
C = CASE
    WHEN A=2 THEN 1
    WHEN A=1 THEN 3
    ELSE 0
END;
...  
```

**Example**

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;
...  
```

**See also**

- `WHILE ... DO, CASE`
- **7.6.6. WHILE ... DO**

**Used for**
Looping constructs

Available in
PSQL

Syntax

```
WHILE <condition> DO
    <compound_statement>
```

Table 85. WHILE … DO Parameters

<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>
<tr>
<td>single_statement</td>
<td>A single statement terminated with a semicolon</td>
</tr>
<tr>
<td>compound_statement</td>
<td>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.

Example

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:

```
S
=====
10
```
See also
IF … THEN … ELSE, LEAVE, EXIT, FOR SELECT, FOR EXECUTE STATEMENT

7.6.7. LEAVE

Used for
Terminating a loop

Available in
PSQL

Syntax

```
[label:]
<loop_stmt>
BEGIN
...
LEAVE [label];
...
END

<loop_stmt> ::= 
    FOR <select_stmt> INTO <var_list> DO
| FOR EXECUTE STATEMENT ... INTO <var_list> DO
| WHILE (<condition>)} DO
```

Table 86. LEAVE 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>

A LEAVE statement immediately terminates the inner loop of a WHILE or FOR looping statement. The label parameter is optional.

LEAVE can cause an exit from outer loops as well. Code continues to be executed from the first statement after the termination of the outer loop block.

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.
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.

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;
                ELSE
                    SUSPEND;
                END
            END
        END
    END
LOOPA:
    ...

See also
EXIT
7.6.8. EXIT

*Used for*
Terminating module execution

*Available in*
PSQL

**Syntax**

EXIT

The `EXIT` statement causes execution of the procedure or trigger to jump to the final `END` statement from any point in the code, thus terminating the program.

**Example**
Using the `EXIT` statement in a selectable procedure:

```
CREATE PROCEDURE GEN_100
RETURNS (  
    I INTEGER  
) AS BEGIN  
    I = 1;  
    WHILE (1=1) DO BEGIN  
        SUSPEND;  
        IF (I=100) THEN  
            EXIT;  
        I = I + 1;  
    END  
END
```

*See also*
LEAVE, SUSPEND

7.6.9. 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. Applications using interfaces that wrap the API perform the fetches from selectable procedures transparently.

2. When a SUSPEND statement is executed in an executable stored procedure, it is the same as executing the EXIT statement, resulting in immediate termination of the procedure.

3. 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.

**Example**

Using the SUSPEND statement in a selectable procedure:

```
CREATE PROCEDURE GEN_100
RETURNS (I INTEGER)
AS
BEGIN
    I = 1;
    WHILE (1=1) DO
        BEGIN
            SUSPEND;
            IF (I=100) THEN
                EXIT;
            I = I + 1;
        END
    END
END
```

**See also**

EXIT

**7.6.10. EXECUTE STATEMENT**

**Used for**

Executing dynamically created SQL statements
Available in
PSQL

Syntax

```sql
<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] <connect_string>

<connect_string> ::= [<hostspec>] {filepath | db_alias}

<hostspec> ::= <tcpip_hostspec> | <NamedPipes_hostspec>

<tcpip_hostspec> ::= hostname[/port]:

<NamedPipes_hostspec> ::= \\

<variables> ::= [:]varname [, [:]varname ...]
```

Table 87. 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>User name. 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>
</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.

**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
    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)
                    (current_driver, current_location)
                    ON EXTERNAL connect_string
                    INTO license_num;
                    ...

WITH {AUTONOMOUS | COMMON} TRANSACTION

Traditionally, the executed SQL statement always ran within the current transaction, and this is still
the default. WITH AUTONOMOUS TRANSACTION causes a separate transaction to be started, with the same
parameters as the current transaction. It will be committed if the statement runs to completion
without errors and rolled back otherwise. WITH COMMON TRANSACTION uses the current transaction if
possible.

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 adds to this the privileges of the calling procedure or trigger, 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 the connect 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 the connect 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

• Notice that statements 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 un-committed 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 log-in and whether a separate connection is open depend 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 the connect string). No defaults are used for missing parameters
  ◦ If all three are absent and the connect string contains no hostname, then the new connection is established on the local host 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 the connect 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 user name (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

Notice

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 launched 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.
FOR EXECUTE STATEMENT

7.6.11. FOR SELECT

*Used for*
Looping row-by-row through a selected result set

*Available in*
PSQL

*Syntax*

```
FOR <select_stmt> [AS CURSOR cursorname]
    DO <compound_statement>
```

**Table 88. FOR SELECT Statement Parameters**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>select_stmt</td>
<td>SELECT statement</td>
</tr>
<tr>
<td>cursorname</td>
<td>Cursor name. It must be unique among cursor names in the PSQL module (stored procedure, trigger or PSQL block)</td>
</tr>
<tr>
<td>single_statement</td>
<td>A single statement, terminated with a colon, that performs all the processing for this FOR loop</td>
</tr>
<tr>
<td>compound_statement</td>
<td>A block of statements wrapped in BEGIN…END, that performs all the processing for this FOR loop</td>
</tr>
</tbody>
</table>

*A FOR SELECT statement*

- retrieves each row sequentially from the result set and executes the statement or block of statements on the 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 carry 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 that is located at the end of the SELECT … FROM … specification. In each iteration of the loop, the field values in 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 LEAVE 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 iteration.

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 body code, 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
END
```

2. Nested FOR SELECT loop:
CREATE PROCEDURE RELFIELDS
RETURNS ( 
    RELATION CHAR(32), 
    POS INTEGER, 
    FIELD CHAR(32))
AS 
BEGIN 
    FOR SELECT RDB$RELATION_NAME 
        FROM RDB$RELATIONS 
        ORDER BY 1 
        INTO :RELATION 
    DO 
    BEGIN 
        FOR SELECT 
            RDB$FIELD_POSITION + 1, 
            RDB$FIELD_NAME 
        FROM RDB$RELATION_FIELDS 
        WHERE 
            RDB$RELATION_NAME = :RELATION 
        ORDER BY RDB$FIELD_POSITION 
        INTO :POS, :FIELD 
        DO 
        BEGIN 
            IF (POS = 2) THEN 
                RELATION = ' "';
                SUSPEND;
                END 
        END 
    END 
END 

3. Using the AS CURSOR clause to surface a cursor for the positioned delete of a record:
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

See also
DECLARE CURSOR, LEAVE, SELECT, UPDATE, DELETE

7.6.12. FOR EXECUTE STATEMENT

Used for
Executing dynamically created SQL statements that return a row set

Available in
PSQL

Syntax

FOR <execute_statement> DO <compound_statement>

Table 89. FOR EXECUTE STATEMENT Statement Parameters

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute_stmt</td>
<td>An EXECUTE STATEMENT statement</td>
</tr>
<tr>
<td>single_statement</td>
<td>A single statement, terminated with a colon, that performs all the processing for this FOR loop</td>
</tr>
<tr>
<td>compound_statement</td>
<td>A block of statements wrapped in BEGIN...END, that 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.
Example

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**

### 7.6.13. OPEN

*Used for*

Opening a declared cursor

*Available in*

PSQL

**Syntax**

```sql
OPEN cursorname
```

**Table 90. OPEN Statement Parameter**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursorname</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 the SELECT statement declared for it 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 VARIABLE statement.
If the `SELECT` statement declared for 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.

**Examples**

1. Using the `OPEN` statement:

```sql
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
      FETCH C INTO :RNAME;
      IF (ROW_COUNT = 0) THEN
        LEAVE;
      SUSPEND;
    END
  CLOSE C;
END^;

SET TERM ;^;
```

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 shall 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
            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
                END
                CLOSE CUR_F;
            END
            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
    END
CLOSE CUR_R;
END
7.6.14. FETCH

Used for
Fetching successive records from a data set retrieved by a cursor

Available in
PSQL

Syntax

\[
\text{FETCH ~cursorname~ INTO [[:varname [, [:varname ...]]]}
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursorname</td>
<td>Cursor name. A cursor with this name must be previously declared with a \text{DECLARE CURSOR} statement and opened by an \text{OPEN} statement.</td>
</tr>
<tr>
<td>varname</td>
<td>Variable name</td>
</tr>
</tbody>
</table>

A 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 \text{DECLARE CURSOR} statement.

The INTO clause gets data from the current row of the cursor and loads them into PSQL variables.

For checking whether all of the data set rows have been fetched, the context variable \text{ROW_COUNT} returns the number of rows fetched by the statement. It is positive until all rows have been checked. A \text{ROW_COUNT} of 1 indicates that the next fetch will be the last.

Example

Using the FETCH 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
         FETCH C INTO :RNAME;
         IF (ROW_COUNT = 0) THEN
            LEAVE;
         SUSPEND;
      END
   CLOSE C;
END^ 

SET TERM ;^ 

See also

DECLARE CURSOR, OPEN, CLOSE

7.6.15. close

Used for
Closing a declared cursor

Available in
PSQL

Syntax

CLOSE cursorname

Table 92. close Statement Parameter

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursorname</td>
<td>Cursor name. A cursor with this name must be previously declared with a</td>
</tr>
<tr>
<td></td>
<td>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.
Example
Using the CLOSE 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  
            FETCH C INTO :RNAME;
            IF (ROW_COUNT = 0) THEN  
                LEAVE;
            SUSPEND;
        END  
    CLOSE C;
END^
```

See also
DECLARE CURSOR, OPEN, FETCH

7.6.16. 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 93. IN AUTONOMOUS TRANSACTION Statement Parameter

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compound_statement</td>
<td>A statement or a block of statements</td>
</tr>
</tbody>
</table>

An 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. It might be needed
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 made changes being cancelled. If the code executes successfully, the autonomous transaction will be committed.

**Example**

Using an autonomous transaction in a trigger for the database `ON CONNECT` event, in order to log all connection attempts, including those that failed:

```sql
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 (CURRENT_USER IN (SELECT
    USERNAME
    FROM
    BLOCKED_USERS)) THEN
    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
END
```

**See also**

Transaction Control

**7.6.17. POST_EVENT**

*Used for*

Notifying listening clients about database events in a module

*Available in*

PSQL
Syntax

```
POST_EVENT event_name
```

Table 94. 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.

**Example**

Notifying the listening applications about inserting a record into the SALES table:

```
SET TERM ^;
CREATE TRIGGER POST_NEW_ORDER FOR SALES
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
  POST_EVENT 'new_order';
END^;
SET TERM ;^
```

7.7. Trapping and Handling Errors

Firebird has a useful lexicon of PSQL statements and resources for trapping errors in modules and for handling them. Internally-implemented exceptions exist for stalling execution when every sort of standard error occurs in DDL, DSQL and the physical environment.

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 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.
7.7.1. 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"

7.7.2. Custom Exceptions

Custom exceptions can be declared in the database as persistent objects and called in the PSQL code to signal specific errors; for instance, to enforce certain business rules. A custom exception consists of an identifier and a default message of approximately 1000 bytes. For details, see CREATE EXCEPTION.

7.7.3. EXCEPTION

*Used for*

Throwing a user-defined exception or re-throwing an exception

*Available in*

PSQL

*Syntax*

```sql
EXCEPTION [exception_name [custom_message]]
```

*Table 95. 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>
</tbody>
</table>

An EXCEPTION statement 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 exception can be handled in the statement, by just leaving it with no specific WHEN ... DO handler and allowing the trigger or stored procedure to terminate and roll back all operations. The calling application gets the alternative message text, if any was specified; otherwise, it receives the message originally defined for that exception.

Within the exception-handling block — and only within it — the caught exception can be re-thrown
by executing the EXCEPTION statement without parameters. If located outside the block, the re-thrown EXCEPTION call has no effect.

Custom exceptions are stored in the system table RDB$EXCEPTIONS.

**Examples**

1. Throwing an exception with dynamically generated text:

   ```sql
   ... 
   EXCEPTION EX_BAD_TYPE
       'Incorrect record type with id ' || new.id;
   ... 
   ```

2. Throwing an exception upon a condition in the 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
   ```

3. Throwing an exception upon a condition and replacing the original message with an alternative message:
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

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',
                GDSCODE,
                SQLCODE,
                SQLSTATE);
        -- Re-throw exception
        EXCEPTION;
    END
END

See also
CREATE EXCEPTION, WHEN ... DO

7.7.4. 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
    | GDSCODE errcode }

Table 96. WHEN ... DO Statement Parameters
The `WHEN … DO` statement is used to handle 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 handle 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_not_found` for GDS error 335544551.

After the `DO` clause, another `GDSCODE` context variable, containing the numeric code, becomes available for use in the statement or the block of statements that code the error handler. That numeric code is required if you want to compare a `GDSCODE` exception with a targeted error.

The `WHEN … DO` statement or block is never executed unless one of the events targeted by its conditions occurs in run-time. If the 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 them.

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 ensues. 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 ^;
```
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, GDS_CODE, SQL_CODE, SQL_STATE) 
            VALUES ('ADD_COUNTRY', GDS_CODE, SQL_CODE, SQLSTATE); 
        -- Re-throw exception 
        EXCEPTION; 
    END 
END
```

3. Handling several errors in one WHEN block

```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 
... 
```

See also

EXPECTATION, CREATE EXCEPTION, SQLCODE and GDSCODE Error Codes and Message Texts and SQLSTATE Codes and Message Texts
Chapter 8. Built-in Functions

Upgraders: PLEASE READ!

A large number of 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()

RDB$GET_CONTEXT and its counterpart RDB$SET_CONTEXT are actually predeclared UDFs. They are listed here as internal functions because they are always present — the user doesn’t have to do anything to make them available.

Available in
DSQL, PSQL * As a declared UDF it should be available in ESQL

Syntax

RDB$GET_CONTEXT ('<namespace>', <varname>)

<namespace> ::= SYSTEM | USER_SESSION | USER_TRANSACTION
<varname>   ::= A case-sensitive quoted string of max. 80 characters

Table 97. 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>

Result type
VARCHAR(255)

Description
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. It contains a number of predefined variables, shown below.

**Context variables in the SYSTEM namespace**

**DB_NAME**
- Either the full path to the database or — if connecting via the path is disallowed — its alias.

**NETWORK_PROTOCOL**
- The protocol used for the connection: 'TCPv4', 'WNET', 'XNET' or NULL.

**CLIENT_ADDRESS**
- For TCPv4, this is the IP address. For XNET, the local process ID. For all other protocols this variable is NULL.

**CURRENT_USER**
- Same as global `CURRENT_USER` variable.

**CURRENT_ROLE**
- Same as global `CURRENT_ROLE` variable.

**SESSION_ID**
- Same as global `CURRENT_CONNECTION` variable.

**TRANSACTION_ID**
- Same as global `CURRENT_TRANSACTION` variable.

**ISOLATION_LEVEL**
- The isolation level of the current transaction: 'READ COMMITTED', 'SNAPSHOT' or 'CONSISTENCY'.

**ENGINE_VERSION**
- The Firebird engine (server) version. Added in 2.1.

**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.

**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'))
```

Chapter 8. Built-in Functions

325
8.1.2. RDB$SET_CONTEXT()

RDB$SET_CONTEXT and its counterpart RDB$GET_CONTEXT are actually predeclared UDFs. They are listed here as internal functions because they are always present—the user doesn’t have to do anything to make them available.

Available in

DSQL, PSQL. * As a declared UDF it should be available in ESQL

Syntax

RDB$SET_CONTEXT ('<namespace>', <varname>, <value> | NULL)

<namespace> ::= USER_SESSION | USER_TRANSACTION
<varname>   ::= A case-sensitive quoted string of max. 80 characters
<value>     ::= A value of any type, as long as it's castable to a VARCHAR(255)

Table 98. 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>

Result type

INTEGER

Description

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.
**Examples**

```sql
select rdb$set_context('USER_SESSION', 'MyVar', 493) from rdb$database
```

```sql
rdb$set_context('USER_SESSION', 'RecordsFound', RecCounter);
```

```sql
select rdb$set_context('USER_TRANSACTION', 'Savepoints', 'Yes')
from rdb$database
```

**Notes**

- 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.

**See also**

RDB$GET_CONTEXT()

**8.2. Mathematical Functions**

**8.2.1. ABS()**

*Available in*

DSQL, PSQL

*Possible name conflict*

YES → Read details

**Syntax**

```sql
ABS (number)
```

**Table 99. 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>

**Result type**

Numerical

**Description**

Returns the absolute value of the argument.
8.2.2. ACOS()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Syntax

ACOS (number)

Table 100. 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>

Result type
DOUBLE PRECISION

Description
Returns the arc cosine of the argument.

- The result is an angle in the range [0, pi].
- If the argument is outside the range [-1, 1], NaN is returned.

8.2.3. ASIN()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Syntax

ASIN (number)

Table 101. 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>

Result type
DOUBLE PRECISION

Description
Returns the arc sine of the argument.
The result is an angle in the range [-\(\pi/2\), \(\pi/2\)].
If the argument is outside the range [-1, 1], \(\text{NaN}\) is returned.

### 8.2.4. ATAN()

*Available in*
DSQL, PSQL

*Possible name conflict*
YES → Read details

**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>

**Result type**
DOUBLE PRECISION

*Description*
The function \(\text{ATAN}\) returns the arc tangent of the argument. The result is an angle in the range \(-\pi/2, \pi/2\).

### 8.2.5. ATAN2()

*Available in*
DSQL, PSQL

*Possible name conflict*
YES → Read details

**Syntax**

```
ATAN2 (y, x)
```

<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>

**Result type**
DOUBLE PRECISION
Description
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, 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.

Notes
- A fully equivalent description of this function is the following: 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 ATAN2(θ, θ) is undefined.
- If x is greater than 0, ATAN2(y, x) is the same as ATAN(y/x).
- If both sine and cosine of the angle are already known, ATAN2(sin, cos) gives the angle.

8.2.6. CEIL(), CEILING()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details (Affects CEILING only)

Syntax

```
CEIL[ING] (number)
```

Table 104. 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>

Result type

BIGINT for exact numeric number, or DOUBLE PRECISION for floating point number

Description

Returns the smallest whole number greater than or equal to the argument.

See also

FLOOR()
8.2.7. COS()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Syntax

\[
\text{COS (angle)}
\]

Table 105. 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>

Result type
DOUBLE PRECISION

Description
Returns an angle’s cosine. The argument must be given in radians.

- Any non-NULL result is — obviously — in the range [-1, 1].

8.2.8. COSH()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Syntax

\[
\text{COSH (number)}
\]

Table 106. 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>

Result type
DOUBLE PRECISION

Description
Returns the hyperbolic cosine of the argument.

- Any non-NULL result is in the range [1, INF].
**8.2.9. COT()**

*Available in*
DSQL, PSQL

*Possible name conflict*
YES → [Read details](#)

**Syntax**

```plaintext
COT (angle)
```

**Table 107. 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>

*Result type*
DOUBLE PRECISION

*Description*
Returns an angle’s cotangent. The argument must be given in radians.

**8.2.10. EXP()**

*Available in*
DSQL, PSQL

**Syntax**

```plaintext
EXP (number)
```

**Table 108. 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>

*Result type*
DOUBLE PRECISION

*Description*
Returns the natural exponential, $e^{number}$

*See also*
LN()
**8.2.11. FLOOR()**

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Syntax

\[
\text{FLOOR (number)}
\]

Table 109. 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>

Result type

BIGINT for exact numeric *number*, or DOUBLE PRECISION for floating point *number*

Description

Returns the largest whole number smaller than or equal to the argument.

See also

CEIL(), CEILING()

**8.2.12. LN()**

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Syntax

\[
\text{LN (number)}
\]

Table 110. 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>

Result type

DOUBLE PRECISION

Description

Returns the natural logarithm of the argument.
An error is raised if the argument is negative or 0.

See also
EXP()

8.2.13. LOG()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Syntax

```
LOG (x, y)
```

Table 11.1. 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>

Result type

DOUBLE PRECISION

Description

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.

8.2.14. LOG10()

Available in
DSQL, PSQL

Changed in
2.5

Possible name conflict
YES → Read details
## Syntax

\[
\text{LOG10} \ (\text{number})
\]

<table>
<thead>
<tr>
<th>Table 112. LOG10 Function Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameter</strong></td>
</tr>
<tr>
<td>number</td>
</tr>
</tbody>
</table>

**Result type**

DOUBLE PRECISION

**Description**

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.)

### 8.2.15. MOD()

**Available in**

DSQL, PSQL

**Possible name conflict**

YES → [Read details](#)

## Syntax

\[
\text{MOD} \ (a, \ b)
\]

<table>
<thead>
<tr>
<th>Table 113. MOD Function Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameter</strong></td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>

**Result type**

SMALLINT, INTEGER or BIGINT depending on the type of \( a \). If \( a \) is a floating point type, the result is a BIGINT.

**Description**

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.
8.2.16. PI()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Syntax

PI ()

Result type
DOUBLE PRECISION

Description
Returns an approximation of the value of pi.

8.2.17. POWER()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Syntax

POWER (x, y)

Table 114. 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>

Result type
DOUBLE PRECISION

Description
Returns x to the power of y ($x^y$).

8.2.18. RAND()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Syntax

```
RAND()
```

Result type
DOUBLE PRECISION

Description
Returns a random number between 0 and 1.

8.2.19. ROUND()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Syntax

```
ROUND(number [, scale])
```

Table 115. 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>

Result type
INTEGER, (scaled) BIGINT or DOUBLE PRECISION

Description
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.

**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

### 8.2.20. SIGN()

**Available in**

DSQL, PSQL

**Possible name conflict**

YES → [Read details](#)

**Syntax**

```
SIGN (number)
```

**Table 116. SIGN 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>

**Result type**

SMALLINT

**Description**

Returns the sign of the argument: -1, 0 or 1.

### 8.2.21. SIN()

**Available in**

DSQL, PSQL

**Possible name conflict**

YES → [Read details](#)
Syntax

\[
\text{SIN (angle)}
\]

Table 117. SIN 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>

Result type

DOUBLE PRECISION

Description

Returns an angle's sine. The argument must be given in radians.

- Any non-NULL result is — obviously — in the range \([-1, 1]\).

8.2.22. SINH()

Available in

DSQL, PSQL

Possible name conflict

YES → Read details

Syntax

\[
\text{SINH (number)}
\]

Table 118. 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>

Result type

DOUBLE PRECISION

Description

Returns the hyperbolic sine of the argument.

8.2.23. SQRT()

Available in

DSQL, PSQL

Possible name conflict

YES → Read details
Syntax

```
SQRT (number)
```

Table 119. 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>

Result type

DOUBLE PRECISION

Description

Returns the square root of the argument.

- If `number` is negative, an error is raised.

8.2.24. TAN()

Available in

DSQL, PSQL

Possible name conflict

YES → Read details

Syntax

```
TAN (angle)
```

Table 120. TAN 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>

Result type

DOUBLE PRECISION

Description

Returns an angle’s tangent. The argument must be given in radians.

8.2.25. TANH()

Available in

DSQL, PSQL

Possible name conflict

YES → Read details
**Syntax**

```sql
TANH (number)
```

**Table 121. TANH 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>

**Result type**

DOUBLE PRECISION

**Description**

Returns the hyperbolic tangent of the argument.

- Due to rounding, any non-NULL result is in the range \([-1, 1]\) (mathematically, it's \((-1, 1]\)).

---

**8.2.26. TRUNC()**

**Available in**

DSQL, PSQL

**Syntax**

```sql
TRUNC (number [, scale])
```

**Table 122. 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>

**Result type**

INTEGER, (scaled) BIGINT or DOUBLE PRECISION

**Description**

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
Notes

- 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.

8.3. String Functions

8.3.1. ASCII_CHAR()

Available in
DSQL, PSQL

Possible name conflict
YES → Read details

Syntax

```
ASCII_CHAR (code)
```

<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>

Result type
CHAR(1) CHARACTER SET NONE

Description
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

**Syntax**

```
ASCII_VAL (ch)
```

**Table 124. 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>

**Result type**

SMALLINT

**Description**

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

**Syntax**

```
BIT_LENGTH (string)
```

**Table 125. 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>

**Result type**
**INTEGER**

**Description**

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.

**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

**Syntax**

```sql
CHAR_LENGTH (string)
| CHARACTER_LENGTH (string)
```

**Table 126. CHAR[ACTER]_LENGTH Function Parameter**
### Parameter Description

<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>

**Result type**

INTEGER

**Description**

Gives the length in characters of the input string.

**Notes**

- 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.

**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

**Syntax**

```sql
HASH (string)
```
Table 127. 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>

**Result type**

BIGINT

**Description**

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

**Syntax**

```
LEFT (string, length)
```

Table 128. 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>

**Result type**

VARCHAR or BLOB

**Description**

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 `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.

**See also**

RIGHT()
### 8.3.7. LOWER()

**Available in**
- DSQL, ESQL, PSQL

**Possible name conflict**
- YES → Read details below

**Syntax**

```
LOWER (string)
```

**Table 129. 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>

**Result type**
- (VAR)CHAR or BLOB

**Description**

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).

**Example**

```sql
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
### Syntax

```sql
LPAD (str, endlen [, padstr])
```

### Table 130. 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 (&quot; &quot;)</td>
</tr>
</tbody>
</table>

### Result type

VARCHAR or BLOB

### Description

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.

### Examples

```sql
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 'abcdefghijHello'
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

Syntax

OCTET_LENGTH (string)

Table 131. 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>

Result type
INTEGER

Description
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.

Examples

```
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

Syntax

OVERLAY (string PLACING replacement FROM pos [FOR length])

Table 132. 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>
<tr>
<td>length</td>
<td>The number of characters that are to be overwritten</td>
</tr>
</tbody>
</table>

Result type

VARCHAR or BLOB

Description

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.
Examples

overlay ('Goodbye' placing 'Hello' from 2) -- returns 'GHelloe'
overlay ('Goodbye' placing 'Hello' from 5) -- returns 'GoodHello'
overlay ('Goodbye' placing 'Hello' from 8) -- returns 'GoodbyeHello'
overlay ('Goodbye' placing 'Hello' from 20) -- returns 'GoodbyeHello'

overlay ('Goodbye' placing 'Hello' from 2 for 0) -- r. 'GHellooodbye'
overlay ('Goodbye' placing 'Hello' from 2 for 3) -- r. 'GHellooby'
overlay ('Goodbye' placing 'Hello' from 2 for 6) -- r. 'GHello'
overlay ('Goodbye' placing 'Hello' from 2 for 9) -- r. 'GHello'

overlay ('Goodbye' placing '' from 4) -- returns 'Goodbye'
overlay ('Goodbye' placing '' from 4 for 3) -- returns 'Gooe'
overlay ('Goodbye' placing '' from 4 for 20) -- returns 'Goo'

overlay ('' placing 'Hello' from 4) -- returns 'Hello'
overlay ('' placing 'Hello' from 4 for 0) -- returns 'Hello'
overlay ('' placing 'Hello' from 4 for 20) -- returns 'Hello'

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
REPLACE()

8.3.11. POSITION()

Available in
DSQL, PSQL

Syntax

POSITION (substr IN string)
| POSITION (substr, string [, startpos])

Table 133. 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>

Result type
INTEGER

Description
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.

Notes

- 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 subst 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 subst 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.

Examples

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>position ('be' in 'To be or not to be')</td>
<td>4</td>
</tr>
<tr>
<td>position ('be', 'To be or not to be')</td>
<td>4</td>
</tr>
<tr>
<td>position ('be', 'To be or not to be', 4)</td>
<td>4</td>
</tr>
<tr>
<td>position ('be', 'To be or not to be', 8)</td>
<td>17</td>
</tr>
<tr>
<td>position ('be', 'To be or not to be', 18)</td>
<td>0</td>
</tr>
<tr>
<td>position ('be' in 'Alas, poor Yorick!')</td>
<td>0</td>
</tr>
</tbody>
</table>

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

SUBSTRING()

8.3.12. REPLACE()

Available in

DSQL, PSQL

Syntax

```
REPLACE (str, find, repl)
```

Table 134. REPLACE Function Parameters

<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>
</tbody>
</table>
**Parameter** | **Description**
--- | ---
repl | The replacement string

**Result type**
VARCHAR or BLOB

**Description**
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.

**Examples**

```sql
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
replace ('Billy Wilder', 'xyz', null) -- returns NULL (!)
replace ('Billy Wilder', 'xyz', 'abc') -- returns 'Billy Wilder'
replace ('Billy Wilder', '', 'abc') -- returns 'Billy Wilder'
```

⚠️ 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**
OVERLAY(), SUBSTRING(), POSITION(), CHAR_LENGTH(), CHARACTER_LENGTH()

### 8.3.13. REVERSE()

**Available in**
DSQL, PSQL

**Syntax**

```sql
REVERSE (string)
```

**Table 135. REVERSE Function Parameter**
### Parameter Description

<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>

**Result type**

VARCHAR

**Description**

Returns a string backwards.

**Examples**

```sql
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:

```sql
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**

YES → [Read details](#)

**Syntax**

```
RIGHT (string, length)
```

**Table 136. RIGHT 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. Defines the number of characters to return</td>
</tr>
</tbody>
</table>

**Result type**

VARCHAR or BLOB

**Description**

...
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

Changed in

2.5 (backported to 2.1.4)

Possible name conflict

YES → Read details

Syntax

```
RPAD (str, endlen [, padstr])
```

Table 137. RPAD Function Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>str</code></td>
<td>An expression of a string type</td>
</tr>
<tr>
<td><code>endlen</code></td>
<td>Output string length</td>
</tr>
<tr>
<td><code>endlen</code></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>

Result type

VARCHAR or BLOB

Description

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 \texttt{str} is a BLOB, the result is a BLOB. Otherwise, the result is a \texttt{VARCHAR(\text{endlen})}.
• If \texttt{padstr} is given and equals '' (empty string), no padding takes place.
• If \texttt{endlen} is less than the current string length, the string is truncated to \texttt{endlen}, even if \texttt{padstr} is the empty string.

\begin{center}
\textbf{In Firebird 2.1-2.1.3, all non-BLOB results were of type \texttt{VARCHAR(32765)}, which made it advisable to cast them to a more modest size. This is no longer the case.}
\end{center}

\textbf{Examples}

\begin{verbatim}
   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'
\end{verbatim}

\begin{center}
\textbf{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.}
\end{center}

\textit{See also} \texttt{LPAD()}

\textbf{8.3.16. \texttt{SUBSTRING()}}

\textit{Available in}

DSQL, PSQL

\textit{Changed in}

2.5.1

\textbf{Syntax}

\begin{verbatim}
   SUBSTRING (str FROM startpos [FOR length])
\end{verbatim}

\textit{Table 138. \texttt{SUBSTRING} Function Parameters}

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Parameter} & \textbf{Description} \\
\hline
\texttt{str} & An expression of a string type \\
\texttt{startpos} & Integer expression, the position from which to start retrieving the substring \\
\hline
\end{tabular}
\end{table}
Parameter Description
length The number of characters to retrieve after the startpos

Result types
VARCHAR or BLOB

Description
Returns a string’s substring starting at the given position, either to the end of the string or with a given length.

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.

In Firebird 1.x, startpos and length must be integer literals. In 2.0 and above they can be any valid integer expression.

Starting with Firebird 2.1, this function fully supports binary and text BLOBs of any length and character set. If str is a BLOB, the result is also a BLOB. For any other argument type, the result is a VARCHAR. Previously, the result type used to be CHAR if the argument was a CHAR or a string literal.

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'.

If any argument is NULL, the result is NULL.

Bugs
- If str is a BLOB and the length argument is not present, the output is limited to 32767 characters. Workaround: with long BLOBs, always specify char_length(str)—or a sufficiently high integer—as the third argument, unless you are sure that the requested substring fits within 32767 characters.

  This bug has been fixed in version 2.5.1; the fix was also backported to 2.1.5.

- An older bug in Firebird 2.0, which caused the function to return “false emptystrings” if startpos or length was NULL, was fixed.

Example
insert into AbbrNames(AbbrName)
select substring(LongName from 1 for 3) from LongNames

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.

See also
8.3.17. **TRIM()**

*Available in*

DSQL, PSQL

**Syntax**

```
TRIM ([<adjust>] str)
```

```
<adjust> ::= {[<where>] [what]} FROM
```

```
<where> ::= BOTH | LEADING | TRAILING
```

**Table 139. 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 <em>str</em>. By default it is space (’ ’)</td>
</tr>
</tbody>
</table>

**Result type**

VARCHAR or BLOB

**Description**

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.
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
-- returns 'Lalala I love you El'
```

Notes

- If `str` is a BLOB, the result is a BLOB. Otherwise, it is a VARCHAR(n) with `n` the formal length of `str`.
- The substring to be removed, if specified, may not be bigger than 32767 bytes. However, if this substring is repeated at `str`'s head or tail, the total number of bytes removed may be far greater. (The restriction on the size of the substring will be lifted in Firebird 3.)

⚠️ 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.

8.3.18. UPPER()

Available in
DSQL, ESQL, PSQL

Syntax

```
UPPER (str)
```

Table 140. UPPER Function Parameter

<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>

Result type

(VAR)CHAR or BLOB

Description
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.

Examples

```
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()

Available in
DSQL, PSQL

Changed in
2.5

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 141. 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. A negative value is subtracted</td>
</tr>
<tr>
<td>unit</td>
<td>Date/time unit</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>datetime</td>
<td>An expression of the DATE, TIME or TIMESTAMP type</td>
</tr>
</tbody>
</table>

**Result type**

DATE, TIME or TIMESTAMP

**Description**

Adds the specified number of years, months, weeks, days, hours, minutes, seconds or milliseconds to a date/time value. (The WEEK unit is new in 2.5.)

- The result type is determined by the third argument.
- 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.

**Examples**

```sql
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')
```

**See also**

DATEDIFF(), Operations Using Date and Time Values

### 8.4.2. DATEDIFF()

**Available in**

DSQL, PSQL

**Changed in**

2.5

**Syntax**

```sql
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 142. 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>

Result type
BIGINT

Description
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.

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()
Syntax

```sql
EXTRACT (<part> FROM <datetime>)
```

<part> ::= YEAR | MONTH | WEEK
| DAY | WEEKDAY | YEARDAY
| HOUR | MINUTE | SECOND | MILLISECOND

<datetime> ::= a DATE, TIME or TIMESTAMP expression

### Table 143. 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>

#### Result type

SMALLINT or NUMERIC

#### Description

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 144. 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>MILISECOND</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**

*Description*

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**

*Description*

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

*Changed in*

2.5
Syntax

\[
\text{CAST} \ (<\text{expression}> \ AS \ <\text{target\_type}>)
\]

\(<\text{target\_type}> ::= \ \\
\text{<\text{sql\_datatype}>} \ \\
[\text{TYPE \ OF}] \ \text{domain} \ \\
\text{TYPE \ OF \ COLUMN} \ \text{relname.column}
\]

**Table 145. CAST Function Parameters**

<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>
<tr>
<td>domain</td>
<td></td>
</tr>
<tr>
<td>relname</td>
<td>Table or view name</td>
</tr>
<tr>
<td>colname</td>
<td>Table or view column name</td>
</tr>
</tbody>
</table>

**Result type**

User-chosen.

**Description**

CAST converts an expression to the desired datatype or domain. If the conversion is not possible, an error is raised.

**“Shorthand” Syntax**

Alternative syntax, supported only when casting a string literal to a DATE, TIME or TIMESTAMP:

\[
datatype \ 'date/timestring'
\]

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.

**Cast Examples**

A full-syntax cast:

```
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
```

The following table shows the type conversions possible 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>[VAR]CHAR</td>
<td>[VAR]CHAR</td>
</tr>
<tr>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>Numeric types</td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATE</td>
<td>[VAR]CHAR</td>
</tr>
<tr>
<td></td>
<td>BLOB</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TIME</td>
<td>[VAR]CHAR</td>
</tr>
<tr>
<td></td>
<td>BLOB</td>
</tr>
<tr>
<td></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 input fields**

Since Firebird 2.0, you can cast statement parameters to a datatype:
cast (?) as integer

This gives you control over the type of input field 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$database; ①
select cast (8000 as quint) from rdb$database; ②
select cast (null as quint) from rdb$database; ③
```

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.

```sql
create table ttt (
  s varchar(40) character set utf8 collate unicode_ci_ai
);
commit;

select cast ('Jag har många vänner' as type of column ttt.s) from rdb$database;
```

**Warnings**

- For text types, character set and collation are preserved by the cast—just as when casting to a domain. However, due to a bug, the collation is not always taken into consideration when comparisons (e.g. equality tests) are made. In cases where the collation is of importance, test your code thoroughly before deploying! This bug is fixed for Firebird 3.

- If a column's definition is altered, existing CASTs to that column's 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 BLOBs

Successful casting to and from BLOBs is possible since Firebird 2.1.

## 8.6. Bitwise Functions

### 8.6.1. BIN_AND()

**Available in**

DSQL, PSQL

**Possible name conflict**

YES → Read details

**Syntax**

```
BIN_AND (number, number [, number ...])
```

**Table 147. 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>
**Result type**
SMALLINT, INTEGER or BIGINT

**Description**
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

**Possible name conflict**
NO

**Syntax**

```
BIN_NOT (number)
```

#### Table 148. 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>

**Result type**
SMALLINT, INTEGER or BIGINT

**Description**
Returns the result of the bitwise **NOT** operation on the argument, i.e. ones 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

Syntax

```
BIN_OR (number, number [, number ...])
```

Table 149. 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>

Result type

SMALLINT, INTEGER or BIGINT

![INFO] 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.

Description

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

Syntax

```
BIN_SHL (number, shift)
```

Table 150. 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>

Result type

BIGINT

Description

Returns the first argument bitwise left-shifted by the second argument, i.e. `a << b` or `a \cdot 2^b`.

See also
BIN_SHR()

**8.6.5. BIN_SHR()**

*Available in*
DSQL, PSQL

**Syntax**

\[
\text{BIN_SHR} \ (\text{number, shift})
\]

*Table 151. 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>

**Result type**

BIGINT

*Description*

Returns the first argument bitwise right-shifted by the second argument, i.e. \( a \gg b \) 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

**Syntax**

\[
\text{BIN_XOR} \ (\text{number}, \text{number} [\ , \text{number} \ldots])
\]

*Table 152. 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>

**Result type**
SMALLINT, INTEGER or BIGINT

**Description**

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

**Added in**  
2.5

**Syntax**

```
CHAR_TO_UUID (ascii_uuid)
```

**Table 153. CHAR_TO_UUID Function Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii_uuid</td>
<td>A 36-character representation of UUID. '-' (hyphen) in positions 9, 14, 19 and 24; valid hexadecimal digits in any other positions, e.g. 'A0bF4E45-3029-2a44-D493-4998c9b439A3'</td>
</tr>
</tbody>
</table>

**Result type**  
CHAR(16) CHARACTER SET OCTETS

**Description**

Converts a human-readable 36-char UUID string to the corresponding 16-byte UUID.

**Examples**

```
select char_to_uuid('A0bF4E45-3029-2a44-D493-4998c9b439A3') from rdb$database  
-- returns A0BF4E4530292A44-D493-4998C9B439A3 (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

Syntax

GEN_UUID ()

Result type
CHAR(16) CHARACTER SET OCTETS

Description
Returns a universally unique ID as a 16-byte character string.

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

Added in
2.5

Syntax

UUID_TO_CHAR (uuid)

Table 154. 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>

Result type
CHAR(36)

Description
Converts a 16-byte UUID to its 36-character, human-readable ASCII representation.

**Examples**

```sql
select uuid_to_char(876C45F4-569B-320D-BCB4-735AC3509E5F) from rdb$database
-- returns '876C45F4-569B-320D-BCB4-735AC3509E5F'

select uuid_to_char(gen_uuid()) from rdb$database
-- returns e.g. '680D946B-45FF-DB4E-B103-BB5711529B86'

select uuid_to_char('Firebird swings!') from rdb$database
-- returns '46697265-6269-7264-2073-77696E677321'
```

**See also**

`CHAR_TO_UUID()`, `GEN_UUID()`

## 8.8. Functions for Sequences (Generators)

### 8.8.1. GEN_ID()

**Available in**

DSQL, ESQL, PSQL

**Syntax**

```sql
GEN_ID (generator-name, step)
```

**Table 155. GEN_ID Function Parameters**

<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>

**Result type**

BIGINT

**Description**

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 `NEXT VALUE FOR` syntax is preferred, except when an increment other than 1 is needed.
Example

\[ \text{new.rec_id} = \text{gen_id}(\text{gen_recnum}, 1); \]

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.

See also

NEXT VALUE FOR, CREATE SEQUENCE (GENERATOR)

8.9. Conditional Functions

8.9.1. COALESCE()

Available in
DSQL, PSQL

Syntax

\[
\text{COALESCE} (\text{<exp1>, <exp2> [, <expN> ... ]})
\]

<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>

Result type

Depends on input.

Description

The \text{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.

Example

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, \text{COALESCE} will happily return that to the caller.

\[
\begin{align*}
\text{select} & \quad \text{coalesce} (\text{Nickname, FirstName, 'Mr./Mrs.'}) \quad | | ' ' | | \text{LastName} \\
& \quad \text{as} \quad \text{FullName} \\
& \quad \text{from} \quad \text{Persons}
\end{align*}
\]
See also
IIF(), NULLIF(), CASE

8.9.2. DECODE()

Available in
DSQL, PSQL

Syntax

\[
\text{DECODE}(\text{testexpr}, \text{expr1}, \text{result1} \\
\text{[expr2}, \text{result2} \ldots] \\
\text{[, defaultresult]})
\]

The equivalent CASE construct:

\[
\text{CASE} \ \text{testexpr} \\
\text{WHEN} \ \text{expr1} \ \text{THEN} \ \text{result1} \\
\text{[WHEN} \ \text{expr2} \ \text{THEN} \ \text{result2} \ \ldots] \\
\text{[ELSE} \ \text{defaultresult}] \\
\text{END}
\]

Table 157. 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>

Result type
Varies

Description
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.
Example

```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

**Syntax**

```
IIF (<condition>, ResultT, ResultF)
```

**Table 158. 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>

**Result type**

Depends on input.

**Description**

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.

**Example**

```sql
select iif( sex = 'M', 'Sir', 'Madam' ) from Customers
```

IIF(<Cond>, Result1, Result2) is a shorthand for “CASE WHEN <Cond> THEN Result1 ELSE Result2 END”.

**See also**
CASE, DECODE()

8.9.4. MAXVALUE()

Available in
DSQL, PSQL

Syntax

```
MAXVALUE (<expr1> [ , ... , <exprN> ])
```

<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>

Result type

Varies according to input — result will be of the same data type as the first expression in the list (expr1).

Description

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.

Example

```
SELECT MAXVALUE(PRICE_1, PRICE_2) AS PRICE
FROM PRICELIST
```

See also
MINVALUE()

8.9.5. MINVALUE()

Available in
DSQL, PSQL

Syntax

```
MINVALUE (<expr1> [ , ... , <exprN> ])
```

<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>

Chapter 8. Built-in Functions
Result type
Varies according to input — result will be of the same data type as the first expression in the list (expr1).

Description
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.

Example

```
SELECT MINVALUE(PRICE_1, PRICE_2) AS PRICE
FROM PRICELIST
```

See also
MAXVALUE()

8.9.6. NULLIF()

Available in
DSQL, PSQL

Syntax

```
NULLIF (<exp1>, <exp2>)
```

Table 161. 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>

Description
NULLIF returns the value of the first argument, unless it is equal to the second. In that case, NULL is returned.

Result type
Depends on input.

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 \(\text{AVG(Weight)}\) would include the -1 weights, thus skewing the result.

See also

\text{COALESCE()}, \text{DECODE()}, \text{IIF()}, \text{CASE}

### 8.10. Aggregate Functions

Aggregate functions operate on groups of records, rather than on individual records or variables. They are often used in combination with a \text{GROUP BY} clause.

#### 8.10.1. \text{AVG()}

**Available in**

DSQL, ESQL, PSQL

**Syntax**

\[
\text{AVG} \left( \left[ \text{ALL} \mid \text{DISTINCT} \right] \text{<expr>} \right)
\]

**Table 162. AVG 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 that returns a numeric data type. Aggregate functions are not allowed as expressions</td>
</tr>
</tbody>
</table>

**Result type**

A numeric data type, the same as the data type of the argument.

**Description**

\(\text{AVG}\) returns the average argument value in the group. \text{NULL} is ignored.

- Parameter \text{ALL} (the default) applies the aggregate function to all values.
- Parameter \text{DISTINCT} directs the \text{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 \text{NULL}, the result will be \text{NULL}.

**Example**

```sql
SELECT
department_id,
\text{AVG(salary)}
FROM employee
GROUP BY department_id
```

See also

SELECT
8.10.2. COUNT()

Available in
DSQL, ESQL, PSQL

Syntax

\[
\text{COUNT ([ALL | DISTINCT] <expr> | *)}
\]

Table 163. 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 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>

Result type

Integer

Description

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 row 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.

Example

```sql
SELECT
department_id,
COUNT(*) AS cnt,
COUNT(DISTINCT name) AS cnt_name
FROM employee
GROUP BY department_id
```

See also

SELECT.
**8.10.3. LIST()**

*Available in*
DSQL, PSQL

*Changed in*
2.5

**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>

**Result type**

BLOB

**Description**

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.

**Examples**

1. Retrieving the list, order undefined:
2. Retrieving the list in alphabetical order, using a derived table:

```sql
SELECT LIST (display_name, '; ') FROM GR_WORK;
```

```sql
SELECT LIST (display_name, '; ') FROM (SELECT display_name FROM GR_WORK ORDER BY display_name);
```

See also

**SELECT**

**8.10.4. MAX()**

*Available in*

DSQL, ESQL, PSQL

*Syntax*

```sql
MAX ([ALL | DISTINCT] <expr>)
```

*Table 165. 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>

*Result type*

Returns a result of the same data type the input expression.

*Description*

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.

ℹ️ The DISTINCT parameter makes no sense if used with MAX() and is implemented only for compliance with the standard.
Example

```sql
SELECT
department,  
MAX(salary)  
FROM employee  
GROUP BY department
```

See also

`MIN()`, `SELECT`

### 8.10.5. `MIN()`

**Available in**

DSQL, ESQL, PSQL

**Syntax**

```sql
MIN ([ALL | DISTINCT] <expr>)
```

**Table 166. MIN Function Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expr</code></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>

**Result type**

Returns a result of the same data type the input expression.

**Description**

`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.

ℹ️ The `DISTINCT` parameter makes no sense if used with `MIN()` and is implemented only for compliance with the standard.
Example

```
SELECT
department,
MIN(salary)
FROM employee
GROUP BY department
```

See also

`MAX()`, `SELECT`

### 8.10.6. SUM()

Available in

DSQL, ESQL, PSQL

#### Syntax

```
SUM ([ALL | DISTINCT] <expr>)
```

#### Table 167. 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>

#### Result type

Returns a result of the same numeric data type as the input expression.

#### Description

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 afterwards.

Example

```
SELECT
department,
SUM (salary),
FROM employee
GROUP BY department
```

See also

`SELECT`
Chapter 9. Context Variables

9.1. CURRENT_CONNECTION

Available in
DSQL, PSQL

Syntax

   CURRENT_CONNECTION

Type
INTEGER

Description
CURRENT_CONNECTION contains the unique identifier of the current connection.

Examples

   select current_connection from rdb$database
   execute procedure P_Login(current_connection)

The value of CURRENT_CONNECTION is a 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.

9.2. CURRENT_DATE

Available in
DSQL, PSQL, ESQL

Syntax

   CURRENT_DATE

Type
DATE

Description
CURRENT_DATE returns the current server date.

Examples

   select current_date from rdb$database
   -- returns e.g. 2011-10-03
Notes

- 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'.

9.3. CURRENT_ROLE

Available in
DSQL, PSQL

Syntax

```
CURRENT_ROLE
```

Type
VARCHAR(31)

Description
CURRENT_ROLE is a context variable containing the role of the currently connected user. If there is no active role, CURRENT_ROLE is 'NONE'.

Example

```
if (current_role <> 'MANAGER')
    then exception only_managers_may_delete;
else
    delete from Customers where custno = :custno;
```

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.

9.4. CURRENT_TIME

Available in
DSQL, PSQL, ESQL

Syntax

```
CURRENT_TIME [ (<precision>) ]
```

```
<precision> ::= 0 | 1 | 2 | 3
```

The optional precision argument is not supported in ESQL.

Table 168. CURRENT_TIME Parameter
CURRENT_TIMESTAMP returns the current server time. In versions prior to 2.0, the fractional part used to be always “.0000”, giving an effective precision of 0 decimals. From Firebird 2.0 onward you can specify a precision when polling this variable. The default is still 0 decimals, i.e. seconds precision.

Examples

```sql
select current_time from rdb$database
-- returns e.g. 14:20:19.6170

select current_time(2) from rdb$database
-- returns e.g. 14:20:23.1200
```

Notes

- Unlike CURRENT_TIME, the default precision of CURRENT_TIMESTAMP has changed to 3 decimals. As a result, CURRENT_TIMESTAMP is no longer the exact sum of CURRENT_DATE and CURRENT_TIME, unless you explicitly specify a precision.
- 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'.

### 9.5. CURRENT_TIMESTAMP

**Available in**

DSQL, PSQL, ESQL

**Syntax**

```
CURRENT_TIMESTAMP [ (<precision>) ]
```

<precision> ::= 0 | 1 | 2 | 3

The optional precision argument is not supported in ESQL.

**Table 169. CURRENT_TIMESTAMP 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>

**Type**
TIMESTAMP

Description

CURRENT_TIMESTAMP returns the current server date and time. In versions prior to 2.0, the fractional part used to be always “.0000”, giving an effective precision of 0 decimals. From Firebird 2.0 onward you can specify a precision when polling this variable. The default is 3 decimals, i.e. milliseconds precision.

Examples

```sql
select current_timestamp from rdb$database
-- returns e.g. 2008-08-13 14:20:19.6170

select current_timestamp(2) from rdb$database
-- returns e.g. 2008-08-13 14:20:23.1200
```

Notes

- The default precision of CURRENT_TIME is still 0 decimals, so in Firebird 2.0 and up CURRENT_TIMESTAMP is no longer the exact sum of CURRENT_DATE and CURRENT_TIME, unless you explicitly specify a precision.
- Within a PSQL module (procedure, trigger or executable block), the value of 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 'NOW'.

9.6. CURRENT_TRANSACTION

Available in
DSQL, PSQL

Syntax

```sql
CURRENT_TRANSACTION
```

Type
INTEGER

Description
CURRENT_TRANSACTION contains the unique identifier of the current transaction.

Examples

```sql
select current_transaction from rdb$database

New.Txn_ID = current_transaction;
```

The value of CURRENT_TRANSACTION is a unique identifier of the current transaction. Its value is
9.7. CURRENT_USER

Available in
DSQL, PSQL

Syntax

CURRENT_USER

Type
VARCHAR(31)

Description
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

9.8. DELETING

Available in
PSQL

Type
boolean

Description
Available in triggers only, DELETING indicates if the trigger fired because of 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
9.9. GDSCODE

Available in
PSQL

Type
INTEGER

Description
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 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.

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 
```

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.

9.10. INSERTING

Available in
PSQL

Type
boolean

Description
Available in triggers only, INSERTING indicates if the trigger fired because of an INSERT operation. Intended for use in multi-action triggers.
Example

if (inserting or updating) then
begin
  if (new.serial_num is null) then
      new.serial_num = gen_id(gen_serials, 1);
end

9.11. NEW

Available in
PSQL, triggers only

Type
Data row

Description
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. But 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.

9.12. 'NOW'

Available in
DSQL, PSQL, ESQL

Changed in
2.0

Type
CHAR(3)

Description
'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. The fractional part of the time used to be always "0.000", giving an effective seconds precision. 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.

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
```

Notes

- '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)`.  

### 9.13. OLD

**Available in**
PSQL, triggers only

**Type**
Data row

**Description**
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. But 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.

### 9.14. ROW_COUNT

*Available in*

PSQL

*Changed in*

2.0

*Type*

INTEGER

*Description*

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.

*Example*

```sql
update Figures set Number = 0 where id = :id;
if (row_count = 0) then
    insert into Figures (id, Number) values (:id, 0);
```

*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.

### 9.15. SQLCODE

*Available in*

PSQL

*Deprecated in*

2.5.1
Type
INTEGER

Description
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 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.

Example

```plaintext
when any
do
begin
  if (sqlcode <> 0) then
    Msg = 'An SQL error occurred!';
  else
    Msg = 'Something bad happened!';
exception ex_custom Msg;
end
```

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.

9.16. SQLSTATE

Available in
PSQL

Added in
2.5.1

Type
CHAR(5)

Description
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.
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
```

Notes

- **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.

9.17. 'TODAY'

Available in
DSQL, PSQL, ESQL

Type
CHAR(5)

Description
'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.
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;
```

Notes

- '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'.

9.18. 'TOMORROW'

Available in
DSQL, PSQL, ESQL

Type
CHAR(8)

Description
'TOMORROW' 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 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;
```

### 9.19. **UPDATING**

*Available in*

PSQL

*Type*

boolean

*Description*

Available in triggers only, **UPDATING** indicates if the trigger fired because of an `UPDATE` 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
```

### 9.20. **'YESTERDAY'**

*Available in*

DSQL, PSQL, ESQL

*Type*

CHAR(9)

*Description*

"'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

```sql
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:

```sql
select date 'Yesterday' from rdb$database;
select timestamp 'YESTERDAY' from rdb$database;
```

### 9.21. USER

**Available in**

DSQL, PSQL

**Syntax**

```sql
USER
```

**Type**

VARCHAR(31)

**Description**

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 10. Transaction Control

Everything in Firebird happens in transactions. Units of work are isolated between a start point and an end point. Changes to data remain reversible until the moment the client application instructs the server to commit them.

10.1. 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

10.1.1. **SET TRANSACTION**

*Used for*
Configuring and starting a transaction

*Available*
DSQL, ESQL
Syntax

```
SET TRANSACTION
  [NAME tr_name]
  [READ WRITE | READ ONLY]
  [[ISOLATION LEVEL]
   { SNAPSHOT [TABLE STABILITY]
     | READ COMMITTED [[NO] RECORD_VERSION] }]
  [WAIT | NO WAIT]
  [LOCK TIMEOUT seconds]
  [NO AUTO UNDO]
  [IGNORE LIMBO]
  [RESERVING <tables> | USING <dbhandles>]

<tables> ::= <table_spec> [, <table_spec> ...]

<table_spec> ::= tablename [, tablename ...]
  [FOR [SHARED | PROTECTED] {READ | WRITE}]

<dbhandles> ::= dbhandle [, dbhandle ...]
```

Table 170. SET TRANSACTION 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>seconds</td>
<td>The time in seconds for the statement to wait in case a conflict occurs</td>
</tr>
<tr>
<td>tables</td>
<td>The list of tables to reserve</td>
</tr>
<tr>
<td>dbhandles</td>
<td>The list of databases the database can access. Available only in ESQL</td>
</tr>
<tr>
<td>table_spec</td>
<td>Table reservation specification</td>
</tr>
<tr>
<td>tablename</td>
<td>The name of the table to reserve</td>
</tr>
<tr>
<td>dbhandle</td>
<td>The handle of the database the database can access. Available only in ESQL</td>
</tr>
</tbody>
</table>

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 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 $2^{31} - 1$, or 2,147,483,647.

All clauses in the `SET TRANSACTION` statement are optional. If the statement starting a transaction has no clauses specified in it, the transaction will be started with default values for access mode, lock resolution mode and isolation level, which are:
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`.

**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`, `TABLE STABILITY`)
- a mechanism for reserving or releasing tables (the `RESERVING` clause)

**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.

**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, it does not apply to global temporary tables (GTT) that are allowed to be changed in `READ ONLY` transactions.

**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.

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 process 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 this one 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.

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 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. 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).

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.

RECORD_VERSION

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 NO 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 unused record versions (garbage) in the event of rollback. With NO AUTO UNDO flagged, the ROLLBACK statement just marks the transaction as rolled back without deleting the unused 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.

**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 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 171. Compatibility of Access Options for RESERVING**

<table>
<thead>
<tr>
<th></th>
<th>SHARED READ</th>
<th>SHARED WRITE</th>
<th>PROTECTED READ</th>
<th>PROTECTED WRITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHARED READ</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SHARED WRITE</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PROTECTED READ</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PROTECTED WRITE</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</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 **SHARED READ** do not affect one other’s access
  - A concurrent mix of **SNAPSHOT** and **READ COMMITTED** transactions with **SHARED 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 SHARED 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 SHARED 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 databases the transaction can access to an enumerated list (of databases). USING is incompatible with RESERVING. A USING clause in SET TRANSACTION syntax is not supported in DSQL.

*See also*

**COMMIT, ROLLBACK**

### 10.1.2. COMMIT

*Used for*

Committing a transaction
Available
DSQL, ESQL

Syntax

```
COMMIT [WORK] [TRANSACTION tr_name]
[RELEASE] [RETAIN [SNAPSHOT]];  
```

**Table 172. 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.

**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 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 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.

*See also*

**SET TRANSACTION, ROLLBACK**

### 10.1.3. ROLLBACK

**Used for**

Rolling back a transaction

**Available**

DSQL, ESQL

**Syntax**

```
ROLLBACK [WORK] [TRANSACTION tr_name]
[RETAIN [SNAPSHOT] | TO [SAVEPOINT] sp_name | RELEASE]
```

**Table 173. 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.

**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 of the 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 optional TO SAVEPOINT clause in the ROLLBACK 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 created 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 are released. Other transactions that have not already requested the rows can request and access the unlocked rows immediately.

See also

SAVEPOINT
10.1.4. SAVEPOINT

**Used for**

Creating a savepoint

**Available**

DSQL

**Syntax**

```
SAVEPOINT sp_name
```

**Table 174. SAVEPOINT Statement Parameter**

<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 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 deleted 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

10.1.5. RELEASE SAVEPOINT

Used for
Erasing a savepoint

Available
DSQL

Syntax

RELEASE SAVEPOINT sp_name [ONLY]

Table 175. RELEASE SAVEPOINT Statement Parameter

<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

10.1.6. 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.

10.1.7. 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 for 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 11. Security

Databases must be secure and so must the data stored in them. Firebird provides two levels of data security protection: user authentication at the server level and SQL privileges within databases. This chapter tells you how to manage security at both levels.

11.1. User Authentication

The security of the entire database depends on identifying a user on verifying its authority, a procedure known as authentication. The information about users authorised to access a specific Firebird server is stored in a special security database named security2.fdb. Each record in security2.fdb is a user account for one user.

A user name, consisting of up to 31 characters, is a case-insensitive system identifier. A user must have a password, of which the first eight are significant. Whilst it is valid to enter a password longer than eight characters, any subsequent characters are ignored. Passwords are case-sensitive.

If the user specified during the connection is the SYSDBA, the database owner or a specially privileged user, that user will have unlimited access to the database.

11.1.1. 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 security2.fdb. No user has access to the metadata of the security database.

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 MacOSX, Firebird will interpret a POSIX user account as though it were a Firebird user account in its own security database, provided the server sees the client machine as a trusted host and the system user accounts exist on both the client and the server. To establish a “trusted” relationship with the client host, the corresponding entries must be included in one of the files /etc/hosts.equiv or /etc/gds_hosts.equiv on Firebird’s host server.

- The file hosts.equiv contains trusted relationships at operating system level, encompassing all services (rlogin, rsh, rcp, and so on)
• The file `gds_hosts.equiv` contains trusted relationships between Firebird hosts only.

The format is identical for both files and looks like this:

```
hostname [username]
```

**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 `security2.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 POSIX host systems. Firebird interprets 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. Trusted Authentication must be enabled by setting the `Authentication` parameter to `Trusted` or `Mixed` in the configuration file, `firebird.conf`.

Even with trusted authentication enabled, Windows operating system Administrators are not automatically granted SYSDBA privileges when they connect 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**.

The embedded version of Firebird server on Windows does not use server-level authentication. However, because objects within a database are subject to SQL privileges, a valid user name and, if applicable, a role, may be required in the connection parameters.

**The Database Owner**

The “owner” of a database is either the user who was CURRENT_USER at the time of creation or, if the parameters `USER` and `PASSWORD` were supplied in the `CREATE DATABASE` statement, the user cited there.

“Owner” is not a user name. The user who is the owner of a database has full administrator rights 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.
11.1.2. RDB$ADMIN Role

The internally-created role RDB$ADMIN is present in every database. Assigning the RDB$ADMIN role to a regular user in a database grants that user the privileges of the SYSDBA, in the current database only.

The elevated privileges take effect when the user is logged in to that regular database under the RDB$ADMIN role and give full control over all objects in the 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, 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:

```sql
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 parameters to the statements CREATE USER and ALTER USER.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_user</td>
<td>Using CREATE USER, name for the new user</td>
</tr>
<tr>
<td>existing_user</td>
<td>Using ALTER USER, Name of an existing user</td>
</tr>
<tr>
<td>password</td>
<td>Using CREATE USER, password for the new user. Its theoretical limit is 31</td>
</tr>
<tr>
<td></td>
<td>bytes but only the first 8 characters are considered.</td>
</tr>
</tbody>
</table>

The grantor must be already logged in as an administrator.

See also

CREATE USER, ALTER USER
Doing the Same Task Using gsec

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$ADMIN role when connecting. No user can connect to the security database, so the solution is that the user connects to a regular database where he also has RDB$ADMIN rights, supplying the RDB$ADMIN role in his login parameters. From there, he can submit any SQL user management command.

The SQL route for the user is blocked for any database in which he has not been the granted the RDB$ADMIN role.

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:

```
GRANT [ROLE] RDB$ADMIN TO username
```
```
REVOKE [ROLE] RDB$ADMIN FROM username
```

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 his RDB$ADMIN privileges, the grantee simply includes the role in the connection attributes when connecting to the database.

AUTO ADMIN MAPPING

In Firebird 2.1, Windows Administrators would automatically receive SYSDBA privileges if trusted authentication was configured for server connections. In Firebird 2.5, it is no longer automatic. The
setting of the AUTO ADMIN MAPPING switch now determines whether Administrators have automatic SYSDBA 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 trusted authentication, and
b. without specifying any role

After a successful “auto admin” connection, the current role is set to RDB$ADMIN.

**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

In regular databases, the status of AUTO ADMIN MAPPING is checked only at connection time. If an Administrator has the RDB$ADMIN role because auto-mapping was on when he logged in, he will keep that role for the duration of the session, even if he 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**

No SQL statements exist to switch automatic mapping on and off in the security database. Instead, `gsec` must be used:

```
gsec -mapping set

gsec -mapping drop
```

More `gsec` switches may be needed, depending on what kind of log-in you used to connect, e.g., `-user` and `-pass`, or `-trusted`.

Only SYSDBA can set the auto-mapping on if it is disabled. Any administrator can drop (disable) it.
11.1.3. 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 177. Administrator (“Superuser”) Characteristics

<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 access to the security database</td>
</tr>
<tr>
<td>root user on POSIX</td>
<td>Auto</td>
<td>Exactly like SYSDBA</td>
</tr>
<tr>
<td>Superuser on POSIX</td>
<td>Auto</td>
<td>Exactly like SYSDBA</td>
</tr>
<tr>
<td>Windows Administrator</td>
<td>Set as CURRENT_ROLE if login succeeds</td>
<td>Exactly like SYSDBA if all of the following are true: In firebird.conf file Authentication = mixed or trusted and Firebird is restarted before proceeding AUTO ADMIN MAPPING Enabled in all databases in which the user needs Superuser privileges Login Does not include a role</td>
</tr>
<tr>
<td>Database owner</td>
<td>Auto</td>
<td>Like SYSDBA, but only in the database of which he is the owner</td>
</tr>
<tr>
<td>Regular user</td>
<td>Must be previously granted; must be supplied at login</td>
<td>Like SYSDBA, but only in the database(s) where the role is granted</td>
</tr>
<tr>
<td>POSIX OS user</td>
<td>Must be previously granted; must be supplied at login</td>
<td>Like SYSDBA, but only in the database(s) where the role is granted</td>
</tr>
<tr>
<td>Windows user</td>
<td>Must be previously granted; must be supplied at login</td>
<td>Like SYSDBA, but only in the database(s) where the role is granted. Not available if config file parameter Authentication = native</td>
</tr>
</tbody>
</table>

11.1.4. SQL Statements for User Management

In Firebird 2.5 and above, user accounts are created, modified and deleted using a series of SQL statements that can be submitted by a user with full administrator rights in the security database.
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 ALTER USER statement and then only to edit some data in their own accounts.

**CREATE USER**

*Used for*
Creating a Firebird user account

*Available in*
DSQL

*Syntax*

```plaintext
CREATE USER username PASSWORD 'password'
  [FIRSTNAME 'firstname']
  [MIDDLENAME 'middlename']
  [LASTNAME 'lastname']
  [GRANT ADMIN ROLE]
```

*Table 178. CREATE USER Statement Parameters*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>User name. The maximum length is 31 characters, following the rules for Firebird regular identifiers. It is always case-insensitive</td>
</tr>
<tr>
<td>password</td>
<td>User password. Its theoretical limit is 31 bytes but only the first 8 characters are considered. Case-sensitive</td>
</tr>
<tr>
<td>firstname</td>
<td>Optional: User's first name. Maximum length 31 characters</td>
</tr>
<tr>
<td>middlename</td>
<td>Optional: User's middle name. Maximum length 31 characters</td>
</tr>
<tr>
<td>lastname</td>
<td>Optional: User's last name. Maximum length 31 characters</td>
</tr>
</tbody>
</table>

Use a CREATE USER statement to create a new Firebird user account. The user must not already exist in the Firebird security database, or a primary key violation error message will be returned.

The `username` argument must follow the rules for Firebird regular identifiers: see Identifiers in the Structure chapter. User names are always case-insensitive. Supplying a user name enclosed in double quotes will not cause an exception: the quotes will be ignored. If a space is the only illegal character supplied, the user name will be truncated back to the first space character. Other illegal characters will cause an exception.

The PASSWORD clause specifies the user's password. A password of more than eight characters is accepted with a warning but any surplus characters will be ignored.
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 (`security2.fdb`). It allows the new user to manage user accounts from any regular database he logs into, but it does not grant the user any special privileges on objects in those databases.

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/ALTER/DROP USER are DDL statements. 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.

**Examples**

1. Creating a user with the username `bigshot`:

   ```sql
   CREATE USER bigshot PASSWORD 'buckshot';
   ```

2. Creating the user `john` with additional properties (first and last names):

   ```sql
   CREATE USER john PASSWORD 'fYe_3Ksw'
   FIRSTNAME 'John'
   LASTNAME 'Doe';
   ```

3. Creating the user `superuser` with user management privileges:

   ```sql
   CREATE USER superuser PASSWORD 'kMn8Kjh'
   GRANT ADMIN ROLE;
   ```

**See also**

ALTER USER, DROP USER

**Used for**

Modifying a Firebird user account

**Available in**

DSQL
Use an ALTER USER statement to edit the details in the named Firebird user account. To modify the account of another user, the current user must have administrator privileges in the security database. Administrator privileges only in regular databases are not sufficient.

Any user can alter his or her own account, except that only an administrator may use GRANT/REVOKE ADMIN ROLE.

All of the arguments are optional but at least one of them must be present:

- The PASSWORD parameter is for specifying a new password for the user
- FIRSTNAME, MIDDLENAME and LASTNAME allow updating of the optional user properties, such as the person's first name, middle name and last name respectively
- Including the clause GRANT ADMIN ROLE grants the user the privileges of the RDB$ADMIN role in the security database (security2.fdb), enabling him/her to manage the accounts of other users. It does not grant the user any special privileges in regular databases.
- Including the clause 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 his or her own

Remember to commit your work if you are working in an application that does not auto-commit DDL.

**Examples**

1. Changing the password for the user bobby and granting him user management privileges:
ALTER USER bobby PASSWORD '67-Uit_68'
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

DROP USER

Used for
Deleting a Firebird user account

Available in
DSQL

Syntax

DROP USER username

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>User name</td>
</tr>
</tbody>
</table>

Table 180. DROP USER Statement Parameter

Use the statement DROP USER to delete a Firebird user account. The current user requires administrator privileges.

Remember to commit your work if you are working in an application that does not auto-commit DDL.

Example
Deleting the user bobby:

DROP USER bobby;
11.2. 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 he 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 he needs 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 user (user, procedure, trigger, role) to which it is granted. Various means are available to grant multiple types of access on an object to multiple users in a single GRANT statement. Privileges may be withdrawn from a user with REVOKE statements.

Privileges are stored in the database to which they apply and are not applicable to any other database.

11.2.1. The Object Owner

The user who creates 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.

Some Ownership Drawbacks

Any authenticated user can access any database and create any valid database object. Up to and including this release, the issue is not controlled.

Because not all database objects are associated with an owner — domains, external functions (UDFs), BLOB filters, generators (sequences) and exceptions — ownerless objects must be regarded as vulnerable on a server that is not adequately protected.

SYSDBA, 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 of the general form:

```
GRANT <privilege> ON <object-type> object-name
    TO { user-name | ROLE role-name }

REVOKE <privilege> ON <OBJECT-TYPE> object-name
    FROM { user-name | ROLE role-name }
```

The object-type is not required for every type of privilege. For some types of privilege, extra
parameters are available, either as options or as requirements.

### 11.2.2. Statements for Granting Privileges

A `GRANT` statement is used for granting privileges—including roles—to users and other database objects.

**GRANT**

*Used for*

Granting privileges and assigning roles

*Available in*

DSQL, ESQL

**Syntax**

```sql
GRANT { <privileges> ON [TABLE] {tablename | viewname} \\
| EXECUTE ON PROCEDURE procname } \\
TO <grantee_list> \\
[WITH GRANT OPTION] \\
[GRANTED BY | AS] [USER] grantor

GRANT <role_granted> \\
TO <role_grantee_list> \\
[WITH ADMIN OPTION] \\
[GRANTED BY | AS] [USER] grantor

<privileges> ::= ALL [PRIVILEGES] | <privilege_list>

<privilege_list> ::= {<privilege> [, <privilege> [, … ] ] } 

<privilege> ::= \\
SELECT | DELETE | INSERT \\
| UPDATE [(col [, col …])] \\
| REFERENCES [(col [, col …])] 

<grantee_list> ::= {<grantee> [, <grantee> [, …] ] }

<grantee> ::= \\
[USER] username | [ROLE] rolename | GROUP Unix_group \\
| PROCEDURE procname | TRIGGER trigname | VIEW viewname | PUBLIC

<role_granted> ::= rolename [, rolename …]

<role_grantee_list> ::= [USER] <role_grantee> [, [USER] <role_grantee> [, …]]

<role_grantee> ::= {username | PUBLIC }
```
### Table 181. GRANT Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tablename</td>
<td>The name of the table the privilege applies to</td>
</tr>
<tr>
<td>viewname</td>
<td>The name of the view the privilege applies to</td>
</tr>
<tr>
<td>procnme</td>
<td>The name of the stored procedure the EXECUTE privilege applies to; or the name of the procedure to be granted the privilege[s]</td>
</tr>
<tr>
<td>col</td>
<td>The table column the privilege is to apply to</td>
</tr>
<tr>
<td>Unix_group</td>
<td>The name of a user group in a POSIX operating system</td>
</tr>
<tr>
<td>username</td>
<td>The user name to which the privileges are granted to or to which the role is assigned</td>
</tr>
<tr>
<td>rolename</td>
<td>Role name</td>
</tr>
<tr>
<td>trigname</td>
<td>Trigger name</td>
</tr>
<tr>
<td>grantor</td>
<td>The user granting the privilege[s]</td>
</tr>
</tbody>
</table>

A GRANT statement grants one or more privileges on database objects to users, roles, stored procedures, triggers or views.

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 the user who has created it (the owner) and administrators have privileges for 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.

**The TO Clause**

The TO clause is used for listing the users, roles and database objects (procedures, triggers and views) that are to be granted the privileges enumerated in privileges. The clause is mandatory.

The optional USER and ROLE keywords 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 checks for a role with this name and, if there is none, the privileges are granted to the user without further checking.

**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 in the DDL chapter for the syntax and rules. The role is maintained by granting privileges to it and, when required, revoking privileges from it. If 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.

Please note:

- 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, such as a procedure, trigger or view, you must specify the object type between the keyword TO and the object name.

- Although the USER and ROLE keywords are optional, it is advisable to use them, in order to avoid ambiguity.

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 user that has been authenticated at login.

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.

If the REVOKE statement is used, it will fail if the current user is not the user that was named in the GRANTED BY clause.

The clauses GRANTED BY and AS can be used only by the database owner and administrators. The object owner cannot use it unless he also has 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

In theory, one GRANT statement grants one privilege to one user or object. In practice, the syntax allows multiple privileges to be granted to multiple users in one GRANT statement.

Syntax extract

```
... 
<privileges> ::= ALL [PRIVILEGES] | <privilege_list>

<privilege_list> ::= {<privilege> [, <privilege> [, … ] ] }

<privilege> ::= 
  SELECT
  | DELETE
  | INSERT
  | UPDATE [([col [, col …]])]
  | REFERENCES [([col [, col …]])]
```

Table 182. List of Privileges on Tables

<table>
<thead>
<tr>
<th>Privilege</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>Permits the user or object to SELECT data from the table or view</td>
</tr>
<tr>
<td>INSERT</td>
<td>Permits the user or object to INSERT rows into the table or view</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Permits the user or object to UPDATE rows in the table or view, optionally restricted to specific columns</td>
</tr>
<tr>
<td>col</td>
<td>(Optional) name of a column to which the user's UPDATE or REFERENCES privilege is restricted</td>
</tr>
<tr>
<td>DELETE</td>
<td>Permits the user or object to DELETE rows from the table or view</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>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.</td>
</tr>
<tr>
<td>ALL</td>
<td>Combines SELECT, INSERT, UPDATE, DELETE and REFERENCES privileges in a single package</td>
</tr>
</tbody>
</table>

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. It allows the grantee to execute the stored procedure 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.

Example

Granting the EXECUTE privilege on a stored procedure to a role:
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:

   GRANT DIRECTOR, MANAGER TO USER IVAN;

2. Assigning the ADMIN 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

11.2.3. Statements for Revoking Privileges

A REVOKE statement is used for revoking privileges—including roles—from users and other database objects.

REVOKE

Used for
Revoking privileges or role assignments

Available in
DSQL, ESQL
Syntax

REVOKE [GRANT OPTION FOR]
  { <privileges> ON [TABLE] {tablename | viewname} |
    EXECUTE ON PROCEDURE procname }
FROM <grantee_list>
[ {GRANTED BY | AS} [USER] grantor ]

REVOKE [ADMIN OPTION FOR] <role_granted>
FROM {PUBLIC | <role_grantee_list> }
[ {GRANTED BY | AS} [USER] grantor ]

REVOKE ALL ON ALL FROM <grantee_list>

<privileges> ::= ALL [PRIVILEGES] | <privilege_list>

<privilege_list> ::= {<privilege> [, <privilege> [, … ] ] }

<privilege> ::= SELECT |
DELETE |
INSERT |
UPDATE [{(col [, col …])}]
| REFERENCES [{(col [, col …])}]

<grantee_list> ::= {<grantee> [, <grantee> [, … ] ] }

<grantee>  ::= [USER] username | [ROLE] rolename | GROUP Unix_group |
            PROCEDURE procname | TRIGGER trigname | VIEW viewname | PUBLIC

<role_granted> ::= rolename [, rolename …]

<role_grantee_list> ::= [USER] <role_grantee> [, [USER] <role_grantee> [, … ] ]

<role_grantee> ::= {username | PUBLIC }

Table 183. REVOKE Statement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tablename</td>
<td>The name of the table the privilege is to be revoked from</td>
</tr>
<tr>
<td>viewname</td>
<td>The name of the view the privilege is to be revoked from</td>
</tr>
<tr>
<td>procname</td>
<td>The name of the stored procedure the EXECUTE privilege is to be revoked from; or the name of the procedure that is to have the privilege[s] revoked</td>
</tr>
<tr>
<td>trigname</td>
<td>Trigger name</td>
</tr>
<tr>
<td>col</td>
<td>The table column the privilege is to be revoked from</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>username</td>
<td>The user name from which the privileges are to be revoked from or the role is to be removed from</td>
</tr>
<tr>
<td>rolename</td>
<td>Role name</td>
</tr>
<tr>
<td>Unix_group</td>
<td>The name of a user group in a POSIX operating system</td>
</tr>
<tr>
<td>grantor</td>
<td>The grantor user on whose behalf the the privilege[s] are being revoked</td>
</tr>
</tbody>
</table>

The `REVOKE` statement is used for revoking privileges from users, roles, stored procedures, triggers and views that were granted using the `GRANT` statement. 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 is used to specify the list of users, roles and database objects (procedures, triggers and views) that will have the enumerated privileges revoked. The optional `USER` and `ROLE` keywords 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 checks for a role with this name and, if there is none, the privileges are revoked from the user without further checking.

**Tips**

- Although the `USER` and `ROLE` keywords are optional, it is advisable to use them in order to avoid ambiguity.
- The `GRANT` 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, 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 privileges on the table, view, trigger or stored procedure to other users or to roles. It does not revoke the privilege with which the grant option is associated.

---

**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. To revoke a privilege that was obtained by this method, the current user must be logged in either with full administrative privileges or as the user designated as grantor by that GRANTED BY clause.

The same rule applies if the syntax used in the original GRANT statement used the synonymous AS form to introduce the clause, instead of the standard GRANTED BY form.

**Revoking ALL ON ALL**

If the current user is logged in with full administrator privileges in the database, the statement

```sql
REVOKE ALL ON ALL FROM <grantee_list>
```

can be used to revoke all privileges (including role memberships) on all objects from one or more users and/or roles. All privileges for the user will be removed, regardless of who granted them. It is a quick way to “clear” privileges when access to the database must be blocked for a particular user or role.

If the current user is not logged in as an administrator, the only privileges revoked will be those that were granted originally by that user.

The REVOKE ALL ON ALL statement cannot be used to revoke privileges that have been granted to stored procedures, triggers or views.

The GRANTED BY clause is not supported.

**Examples using REVOKE**

1. Revoking the privileges for reading and inserting into the SALES

   ```sql
   REVOKE SELECT, INSERT ON TABLE SALES FROM USER ALEX;
   ```

2. Revoking the privilege for reading the CUSTOMER table from the MANAGER and ENGINEER roles and from the user IVAN:
REVOKE SELECT ON TABLE CUSTOMER
FROM ROLE MANAGER, ROLE ENGINEER, USER IVAN;

3. Revoking from the ADMINISTRATOR role the authority to grant any privileges on the CUSTOMER table to other users or roles:

REVOKE GRANT OPTION FOR ALL ON TABLE CUSTOMER
FROM ROLE ADMINISTRATOR;

4. Revoking the privilege for reading the COUNTRY table and the authority to reference the NAME column of the COUNTRY table from any user, via the special user PUBLIC:

REVOKE SELECT, REFERENCES (NAME) ON TABLE COUNTRY
FROM PUBLIC;

5. Revoking the privilege for reading the EMPLOYEE table from the user IVAN, that was granted by the user ALEX:

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:

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:

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:

REVOKE EXECUTE ON PROCEDURE ADD_EMP_PROJ
FROM ROLE MANAGER;

9. Revoking the DIRECTOR and MANAGER roles from the user IVAN:

REVOKE DIRECTOR, MANAGER FROM USER IVAN;
10. Revoke from the user ALEX the authority to assign the MANAGER role to other users:

    REVOKE ADMIN OPTION FOR MANAGER FROM USER IVAN;

11. Revoking all privileges (including roles) on all objects from the user IVAN:

    REVOKE ALL ON ALL FROM IVAN;

    After this statement is executed, the user IVAN will have no privileges whatsoever.

See also

GRANT
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$VALID_BLR fields:
The following query will find the modules that depend on a specific table column and report the state of their `RDB$VALID_BLR` fields:

```sql
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 = 'MYDOMAIN')

/* Replace MYDOMAIN with the actual domain name. Use all-caps if the domain was created case-insensitively. Otherwise, use the exact capitalisation. */
```
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 runtime, 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 <proname> 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:
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"
  a. GDSCODEs 335544366 to 335544334
  b. GDSCODEs 335544454 to 336330760
  c. GDSCODEs 335544329 to 335544613
  d. GDSCODEs 335544614 to 335544689

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 184. 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></td>
</tr>
<tr>
<td>00000</td>
<td>Success</td>
</tr>
<tr>
<td>SQLCLASS 01 (Warning)</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>General warning</td>
</tr>
<tr>
<td>01001</td>
<td>Cursor operation conflict</td>
</tr>
<tr>
<td>01002</td>
<td>Disconnect error</td>
</tr>
<tr>
<td>01003</td>
<td>NULL value eliminated in set function</td>
</tr>
<tr>
<td>01004</td>
<td>String data, right-truncated</td>
</tr>
<tr>
<td>01005</td>
<td>Insufficient item descriptor areas</td>
</tr>
<tr>
<td>01006</td>
<td>Privilege not revoked</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>Mapped Message</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>01007</td>
<td>Privilege not granted</td>
</tr>
<tr>
<td>01008</td>
<td>Implicit zero-bit padding</td>
</tr>
<tr>
<td>01100</td>
<td>Statement reset to unprepared</td>
</tr>
<tr>
<td>01101</td>
<td>Ongoing transaction has been committed</td>
</tr>
<tr>
<td>01102</td>
<td>Ongoing transaction has been rolled back</td>
</tr>
</tbody>
</table>

**SQLCLASS 02 (No Data)**

| 02000    | No data found or no rows affected |

**SQLCLASS 07 (Dynamic SQL error)**

| 07000    | Dynamic SQL error                  |
| 07001    | Wrong number of input parameters   |
| 07002    | Wrong number of output parameters  |
| 07003    | Cursor specification cannot be executed |
| 07004    | USING clause required for dynamic parameters |
| 07005    | Prepared statement not a cursor-specification |
| 07006    | Restricted data type attribute violation |
| 07007    | USING clause required for result fields |
| 07008    | Invalid descriptor count            |
| 07009    | Invalid descriptor index            |

**SQLCLASS 08 (Connection Exception)**

| 08001    | Client unable to establish connection |
| 08002    | Connection name in use               |
| 08003    | Connection does not exist            |
| 08004    | Server rejected the connection       |
| 08006    | Connection failure                   |
| 08007    | Transaction resolution unknown       |

**SQLCLASS 0A (Feature Not Supported)**

| 0A000    | Feature Not Supported                |

**SQLCLASS 0B (Invalid Transaction Initiation)**

| 0B000    | Invalid transaction initiation       |

**SQLCLASS 0L (Invalid Grantor)**

| 0L000    | Invalid grantor                      |

**SQLCLASS 0P (Invalid Role Specification)**

<p>| 0P000    | Invalid role specification           |</p>
<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Mapped Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCLASS 0U (Attempt to Assign to Non-Updatable Column)</td>
<td></td>
</tr>
<tr>
<td>0U000</td>
<td>Attempt to assign to non-updatable column</td>
</tr>
<tr>
<td>SQLCLASS 0V (Attempt to Assign to Ordering Column)</td>
<td></td>
</tr>
<tr>
<td>0V000</td>
<td>Attempt to assign to Ordering column</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>SQLSTATE</td>
<td>Mapped Message</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</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>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>SQLSTATE</td>
<td>Mapped Message</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>SQLCLASS 2B</td>
<td>Dependent Privilege Descriptors Still Exist</td>
</tr>
<tr>
<td>2B000</td>
<td>Dependent privilege descriptors still exist</td>
</tr>
<tr>
<td>SQLCLASS 2C</td>
<td>Invalid Character Set Name</td>
</tr>
<tr>
<td>2C000</td>
<td>Invalid character set name</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>
</tbody>
</table>

Appendix B: Exception Codes and Messages
<table>
<thead>
<tr>
<th>SQLSTATE</th>
<th>Mapped Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>3C000</td>
<td>Ambiguous cursor name</td>
</tr>
<tr>
<td>SQLCLASS 3D (Invalid Catalog Name)</td>
<td></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>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>SQLSTATE</td>
<td>Mapped Message</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------------</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>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>
</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 185. SQLCODE and GDSCODE Error Codes and Message Texts (1)

<table>
<thead>
<tr>
<th>SQLCODE</th>
<th>GDSCODE</th>
<th>Symbol</th>
<th>Message Text</th>
</tr>
</thead>
<tbody>
<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>100</td>
<td>335544374</td>
<td>stream_eof</td>
<td>Attempt to fetch past the last record in a record stream</td>
</tr>
<tr>
<td>0</td>
<td>335741039</td>
<td>gfix_opt_SQL_dialect</td>
<td>-sql_dialect</td>
</tr>
<tr>
<td>0</td>
<td>335544875</td>
<td>bad_debug_format</td>
<td>Bad debug info format</td>
</tr>
<tr>
<td>-84</td>
<td>335544554</td>
<td>nonsql_security_rel</td>
<td>Table/procedure has non-SQL security class 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>-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>dsqll_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 @1 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>-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>dsqll_constant_err</td>
<td>Data type for constant unknown</td>
</tr>
<tr>
<td>-104</td>
<td>336003075</td>
<td>dsqll_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>dsqll_invalid_label</td>
<td>Label @1 @2 in the current scope</td>
</tr>
<tr>
<td>-104</td>
<td>336003088</td>
<td>dsqll_datatypes_not_comparable</td>
<td>Datatypes @1are not comparable in expression @2</td>
</tr>
<tr>
<td>-104</td>
<td>335544343</td>
<td>invalid_blr</td>
<td>Invalid request BLR at offset @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>-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>Bad parameter number</td>
</tr>
<tr>
<td>-104</td>
<td>335544440</td>
<td>bad_msg_vec</td>
<td></td>
</tr>
<tr>
<td>-104</td>
<td>335544456</td>
<td>invalid_sdl</td>
<td>Invalid slice description language at offset @1</td>
</tr>
<tr>
<td>-104</td>
<td>335544570</td>
<td>dsq1_command_err</td>
<td>Invalid command</td>
</tr>
<tr>
<td>-104</td>
<td>335544579</td>
<td>dsq1_internal_err</td>
<td>Internal error</td>
</tr>
<tr>
<td>-104</td>
<td>335544590</td>
<td>dsq1_dup_option</td>
<td>Option specified more than once</td>
</tr>
<tr>
<td>-104</td>
<td>335544591</td>
<td>dsq1_tran_err</td>
<td>Unknown transaction option</td>
</tr>
<tr>
<td>-104</td>
<td>335544592</td>
<td>dsq1_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>dsq1_token_unk_err</td>
<td>Token unknown - line @1, column @2</td>
</tr>
<tr>
<td>-104</td>
<td>335544709</td>
<td>dsq1_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>dsq1_column_pos_err</td>
<td>Invalid column position used in the @1 clause</td>
</tr>
<tr>
<td>-104</td>
<td>335544822</td>
<td>dsq1_agg_where_err</td>
<td>Cannot use an aggregate function in a WHERE clause, use HAVING instead</td>
</tr>
<tr>
<td>-104</td>
<td>335544823</td>
<td>dsq1_agg_group_err</td>
<td>Cannot use an aggregate function in a GROUP BY clause</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>-104</td>
<td>335544824</td>
<td>dsql_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>dsql_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>dsql_agg_nested_err</td>
<td>Nested aggregate 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>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>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>336397228</td>
<td>dsq1_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>dsq1_cte_not_a_union</td>
<td>Recursive CTE (@1) must be an UNION</td>
</tr>
<tr>
<td>-104</td>
<td>336397230</td>
<td>dsq1_cte_nonrecurs_after_recurs</td>
<td>CTE '@1' defined non-recursive member after recursive</td>
</tr>
<tr>
<td>-104</td>
<td>336397231</td>
<td>dsq1_cte_wrong_clause</td>
<td>Recursive member of CTE '@1' has @2 clause</td>
</tr>
<tr>
<td>-104</td>
<td>336397232</td>
<td>dsq1_cte_union_all</td>
<td>Recursive members of CTE (@1) must be linked with another members via UNION ALL</td>
</tr>
<tr>
<td>-104</td>
<td>336397233</td>
<td>dsq1_cte_miss_nonrecursive</td>
<td>Non-recursive member is missing in CTE '@1'</td>
</tr>
<tr>
<td>-104</td>
<td>336397234</td>
<td>dsq1_cte_nested_with</td>
<td>WITH clause can’t be nested</td>
</tr>
<tr>
<td>-104</td>
<td>336397235</td>
<td>dsq1_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>dsq1_cte_not_used</td>
<td>CTE &quot;@1&quot; is not used in query</td>
</tr>
<tr>
<td>-105</td>
<td>335544702</td>
<td>like_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>-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>constaint_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>dsq1_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>-170</td>
<td>335544512</td>
<td>prcmismat</td>
<td>Input parameter mismatch for procedure @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>-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>-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>-203</td>
<td>335544708</td>
<td>dyn_fld_ambiguous</td>
<td>Ambiguous column reference</td>
</tr>
<tr>
<td>-204</td>
<td>336003085</td>
<td>dsql_ambiguous_field_name</td>
<td>Ambiguous field name between @1 and @2</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_defined</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 table/procedure 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>-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>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>335544595</td>
<td>dsql_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>dsql_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>dsql_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 - <code>@1</code></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 255</td>
</tr>
<tr>
<td>-204</td>
<td>335544817</td>
<td>bad_limit_param</td>
<td>Invalid parameter to 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 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>-205</td>
<td>33554396</td>
<td>fldnotdef</td>
<td>Column @1 is not defined in table @2</td>
</tr>
<tr>
<td>-205</td>
<td>33554552</td>
<td>grant_fld_notfound</td>
<td>Could not find column for GRANT</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>-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>dsql_field_err</td>
<td>Column unknown</td>
</tr>
<tr>
<td>-206</td>
<td>335544587</td>
<td>dsql_blob_err</td>
<td>Column is not a BLOB</td>
</tr>
<tr>
<td>-206</td>
<td>335544596</td>
<td>dsql_subselect_err</td>
<td>Subselect illegal in this context</td>
</tr>
<tr>
<td>-206</td>
<td>336397208</td>
<td>dsql_line_col_error</td>
<td>At line @1, column @2</td>
</tr>
<tr>
<td>-206</td>
<td>336397209</td>
<td>dsql_unknown_pos</td>
<td>At unknown line and column</td>
</tr>
<tr>
<td>-206</td>
<td>336397210</td>
<td>dsql_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_small2</td>
<td>Log record header too small at offset @1 in log file @1</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>-239</td>
<td>335544694</td>
<td>partition_too_small</td>
<td>Log partition size too small</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>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>-257</td>
<td>335544566</td>
<td>start_cm_for_wal</td>
<td>WAL defined; Cache Manager must be started first</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>-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>dsql_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>-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 gds software consistency check (invalid RDB$CONSTRAINT_TYPE)</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>-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>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>335544669</td>
<td>dsql_count_mismatch</td>
<td>Count of column list and variable list do not match</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>33554427</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>-407</td>
<td>335544435</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>
</tbody>
</table>

Table 186. SQLCODE and GDSCODE Error Codes and Message Texts (2)

<table>
<thead>
<tr>
<th>SQLCODE</th>
<th>GDSCODE</th>
<th>Symbol</th>
<th>Message Text</th>
</tr>
</thead>
<tbody>
<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>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>-501</td>
<td>335544577</td>
<td>dsqql_cursor_close_err</td>
<td>Attempt to reclose a closed cursor</td>
</tr>
<tr>
<td>-502</td>
<td>336003090</td>
<td>dsqql_cursor_redefined</td>
<td>Statement already has a cursor @1 assigned</td>
</tr>
<tr>
<td>-502</td>
<td>336003091</td>
<td>dsqql_cursor_not_found</td>
<td>Cursor @1 is not found in the current context</td>
</tr>
<tr>
<td>-502</td>
<td>336003092</td>
<td>dsqql_cursor_exists</td>
<td>Cursor @1 already exists in the current context</td>
</tr>
<tr>
<td>-502</td>
<td>336003093</td>
<td>dsqql_cursor_rel_ambiguous</td>
<td>Relation @1 is ambiguous in cursor @2</td>
</tr>
<tr>
<td>-502</td>
<td>336003094</td>
<td>dsqql_cursor_rel_not_found</td>
<td>Relation @1 is not found in cursor @2</td>
</tr>
<tr>
<td>-502</td>
<td>336003095</td>
<td>dsqql_cursor_not_open</td>
<td>Cursor is not open</td>
</tr>
<tr>
<td>-502</td>
<td>335544574</td>
<td>dsqql_decl_err</td>
<td>Invalid cursor declaration</td>
</tr>
<tr>
<td>-502</td>
<td>335544576</td>
<td>dsqql_cursor_open_err</td>
<td>Attempt to reopen an open cursor</td>
</tr>
<tr>
<td>-504</td>
<td>336003089</td>
<td>dsqql_cursor_invalid</td>
<td>Empty cursor name is not allowed</td>
</tr>
<tr>
<td>-504</td>
<td>335544572</td>
<td>dsqql_cursor_err</td>
<td>Invalid cursor reference</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>dsqql_cursor_update_err</td>
<td>Cursor @1 is not updatable</td>
</tr>
<tr>
<td>-518</td>
<td>335544582</td>
<td>dsqql_request_err</td>
<td>Request unknown</td>
</tr>
<tr>
<td>-519</td>
<td>335544688</td>
<td>dsqql_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>dsqql_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 by I/O error</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>
</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>-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>-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>335544644</td>
<td>stream_bof</td>
<td>Illegal operation when at beginning of stream</td>
</tr>
<tr>
<td>-597</td>
<td>335544632</td>
<td>dsq1_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>dsq1_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>dsq1_max_arr_dim_exceeded</td>
<td>Array declared with too many dimensions</td>
</tr>
<tr>
<td>-604</td>
<td>335544594</td>
<td>dsq1_arr_range_error</td>
<td>Illegal array dimension range</td>
</tr>
<tr>
<td>-605</td>
<td>335544682</td>
<td>dsq1_field_ref</td>
<td>Inappropriate self-reference of column</td>
</tr>
<tr>
<td>-607</td>
<td>336003074</td>
<td>dsq1_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>dsq1_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>dsq1_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>335544351</td>
<td>no_meta_update</td>
<td>Unsuccessful metadata update</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>335544549</td>
<td>systrig_update</td>
<td>Cannot modify or erase a system trigger</td>
</tr>
<tr>
<td>-607</td>
<td>335544657</td>
<td>dsql_no_blob_array</td>
<td>Array/BLOB/DATE data types not allowed in arithmetic</td>
</tr>
<tr>
<td>-607</td>
<td>335544746</td>
<td>reftableRequires_pk</td>
<td>&quot;REFERENCES table&quot; without &quot;(column)&quot; requires PRIMARY KEY on referenced table</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>UDF @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>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>-615</td>
<td>33554475</td>
<td>relation_lock</td>
<td>Lock on table @1 conflicts with existing lock</td>
</tr>
<tr>
<td>-615</td>
<td>33554476</td>
<td>record_lock</td>
<td>Requested record lock conflicts with existing lock</td>
</tr>
<tr>
<td>-615</td>
<td>33554507</td>
<td>range_in_use</td>
<td>Refresh range number @1 already in use</td>
</tr>
<tr>
<td>-616</td>
<td>33554530</td>
<td>primary_key_ref</td>
<td>Cannot delete PRIMARY KEY being used in FOREIGN KEY definition</td>
</tr>
<tr>
<td>-616</td>
<td>33554539</td>
<td>integ_index_del</td>
<td>Cannot delete index used by an Integrity Constraint</td>
</tr>
<tr>
<td>-616</td>
<td>33554540</td>
<td>integ_index_mod</td>
<td>Cannot modify index used by an Integrity Constraint</td>
</tr>
<tr>
<td>-616</td>
<td>33554541</td>
<td>check_trig_del</td>
<td>Cannot delete trigger used by a CHECK Constraint</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>-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>-637</td>
<td>335544664</td>
<td>dsql_duplicate spec</td>
<td>Duplicate specification of @1- not supported</td>
</tr>
<tr>
<td>-637</td>
<td>336397213</td>
<td>dsql implicit domain name</td>
<td>Implicit domain name @1 not allowed in user created domain</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>-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>-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>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>-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 32K in length</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>336003097</td>
<td>dsq1_feature_not_supported_ods</td>
<td>Feature not supported on ODS version older than @1.@2</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>dsq1_sqlda_err</td>
<td>SQLDA missing or incorrect version, or incorrect number/type of variables</td>
</tr>
<tr>
<td>-804</td>
<td>335544584</td>
<td>dsq1_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>dsq1_function_err</td>
<td>Function unknown</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>-804</td>
<td>335544713</td>
<td>dsql_sqlda_value_err</td>
<td>Incorrect values within SQLDA structure</td>
</tr>
<tr>
<td>-804</td>
<td>336397205</td>
<td>dsql_too_old_odds</td>
<td>ODS versions before ODS@1 are not supported</td>
</tr>
<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>336003079</td>
<td>isc_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>dsql_incompatible_trigger_type</td>
<td>Incompatible trigger type</td>
</tr>
<tr>
<td>-817</td>
<td>336003103</td>
<td>dsql_db_trigger_type_cant_change</td>
<td>Database trigger type can’t be changed</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>
</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>-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>-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>-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>335544473</td>
<td>invalid_bookmark</td>
<td>Invalid bookmark handle</td>
</tr>
<tr>
<td>-824</td>
<td>335544474</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>335544585</td>
<td>dsql_stmt_handle</td>
<td>Invalid statement handle</td>
</tr>
<tr>
<td>-827</td>
<td>335544655</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>336068816</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>336068817</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>336068818</td>
<td>dyn_dtype_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>336068829</td>
<td>max_coll_per_charset</td>
<td>Maximum number of collations per character set exceeded</td>
</tr>
<tr>
<td>-829</td>
<td>336068830</td>
<td>invalid_coll_attr</td>
<td>Invalid collation attributes</td>
</tr>
<tr>
<td>-829</td>
<td>336068852</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>336068853</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>335544616</td>
<td>field_ref_err</td>
<td>Invalid column reference</td>
</tr>
<tr>
<td>-830</td>
<td>335544615</td>
<td>field_aggregate_err</td>
<td>Column used with aggregate</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>-831</td>
<td>335544548</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>335544604</td>
<td>key_field_count_err</td>
<td>FOREIGN KEY column count does not match PRIMARY KEY</td>
</tr>
<tr>
<td>-833</td>
<td>335544606</td>
<td>expression_eval_err</td>
<td>Expression evaluation not supported</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>-834</td>
<td>33554508</td>
<td>range_not_found</td>
<td>Refresh range number @1 not found</td>
</tr>
<tr>
<td>-835</td>
<td>33554649</td>
<td>bad_checksum</td>
<td>Bad checksum</td>
</tr>
<tr>
<td>-836</td>
<td>33554517</td>
<td>except</td>
<td>Exception @1</td>
</tr>
<tr>
<td>-836</td>
<td>33554848</td>
<td>except2</td>
<td>Exception @1</td>
</tr>
<tr>
<td>-837</td>
<td>33554518</td>
<td>cache_restart</td>
<td>Restart shared cache manager</td>
</tr>
<tr>
<td>-838</td>
<td>33554560</td>
<td>shutwarn</td>
<td>Database @1 shutdown in @2 seconds</td>
</tr>
<tr>
<td>-841</td>
<td>33554577</td>
<td>version_err</td>
<td>Too many versions</td>
</tr>
<tr>
<td>-842</td>
<td>33554597</td>
<td>precision_err</td>
<td>Precision must be from 1 to 18</td>
</tr>
<tr>
<td>-842</td>
<td>33554698</td>
<td>scale_nogt</td>
<td>Scale must be between zero and precision</td>
</tr>
<tr>
<td>-842</td>
<td>33554699</td>
<td>expec_short</td>
<td>Short integer expected</td>
</tr>
<tr>
<td>-842</td>
<td>33554700</td>
<td>expec_long</td>
<td>Long integer expected</td>
</tr>
<tr>
<td>-842</td>
<td>33554701</td>
<td>expec_ushort</td>
<td>Unsigned short integer expected</td>
</tr>
<tr>
<td>-842</td>
<td>33554712</td>
<td>expec_positive</td>
<td>Positive value expected</td>
</tr>
<tr>
<td>-901</td>
<td>335740929</td>
<td>gfix_db_name</td>
<td>Database file name (@1) already given</td>
</tr>
<tr>
<td>-901</td>
<td>336330753</td>
<td>gbak_unknown_switch</td>
<td>Found unknown switch</td>
</tr>
<tr>
<td>-901</td>
<td>336920577</td>
<td>gstat_unknown_switch</td>
<td>Found unknown switch</td>
</tr>
<tr>
<td>-901</td>
<td>336986113</td>
<td>fbsvcmgr_bad_am</td>
<td>Wrong value for access mode</td>
</tr>
<tr>
<td>-901</td>
<td>335740930</td>
<td>gfix_invalid_sw</td>
<td>Invalid switch @1</td>
</tr>
<tr>
<td>-901</td>
<td>33554322</td>
<td>bad_dbkey</td>
<td>Invalid database key</td>
</tr>
<tr>
<td>-901</td>
<td>336986114</td>
<td>fbsvcmgr_bad_wm</td>
<td>Wrong value for write mode</td>
</tr>
<tr>
<td>-901</td>
<td>336330754</td>
<td>gbak_page_size_missing</td>
<td>Page size parameter missing</td>
</tr>
<tr>
<td>-901</td>
<td>336920578</td>
<td>gstat_retry</td>
<td>Please retry, giving a database name</td>
</tr>
<tr>
<td>-901</td>
<td>336986115</td>
<td>fbsvcmgr_bad_rs</td>
<td>Wrong value for reserve space</td>
</tr>
<tr>
<td>-901</td>
<td>336920579</td>
<td>gstat_wrong_ods</td>
<td>Wrong ODS version, expected @1, encountered @2</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>336330755</td>
<td>gbak_page_size_toobig</td>
<td>Page size specified (@1) greater than limit (16384 bytes)</td>
</tr>
<tr>
<td>-901</td>
<td>335740932</td>
<td>gfix_incmp_sw</td>
<td>Incompatible switch combination</td>
</tr>
<tr>
<td>-901</td>
<td>336920580</td>
<td>gstat_unexpected_eof</td>
<td>Unexpected end of database file</td>
</tr>
<tr>
<td>-901</td>
<td>336330756</td>
<td>gbak_redir_ouput_missing</td>
<td>Redirect location for output is not specified</td>
</tr>
<tr>
<td>-901</td>
<td>336986116</td>
<td>fbsvcgmgr_info_err</td>
<td>Unknown tag (@1) in info_svr_db_info block after isc_svc_query()</td>
</tr>
<tr>
<td>-901</td>
<td>335740933</td>
<td>gfix_replay_req</td>
<td>Replay log pathname required</td>
</tr>
<tr>
<td>-901</td>
<td>336330757</td>
<td>gbak_switches_conflict</td>
<td>Conflicting switches for backup/restore</td>
</tr>
<tr>
<td>-901</td>
<td>336986117</td>
<td>fbsvcgmgr_query_err</td>
<td>Unknown tag (@1) in isc_svc_query() results</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>335740934</td>
<td>gfix_pgbuf_req</td>
<td>Number of page buffers for cache required</td>
</tr>
<tr>
<td>-901</td>
<td>336986118</td>
<td>fbsvcgmgr_switch_unknown</td>
<td>Unknown switch &quot;@1&quot;</td>
</tr>
<tr>
<td>-901</td>
<td>336330758</td>
<td>gbak_unknown_device</td>
<td>Device type @1 not known</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>335740935</td>
<td>gfix_val_req</td>
<td>Numeric value required</td>
</tr>
<tr>
<td>-901</td>
<td>336330759</td>
<td>gbak_no_protection</td>
<td>Protection is not there yet</td>
</tr>
<tr>
<td>-901</td>
<td>335544328</td>
<td>bad_segstr_handle</td>
<td>Invalid BLOB handle</td>
</tr>
<tr>
<td>-901</td>
<td>335740936</td>
<td>gfix_pval_req</td>
<td>Positive numeric value required</td>
</tr>
<tr>
<td>-901</td>
<td>336330760</td>
<td>gbak_page_size_not_allowed</td>
<td>Page size is allowed only on restore or create</td>
</tr>
</tbody>
</table>

Table 187. SQLCODE and GDSCODE Error Codes and Message Texts (3)

<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>335544329</td>
<td>bad_segstr_id</td>
<td>Invalid BLOB ID</td>
</tr>
<tr>
<td>-901</td>
<td>335740937</td>
<td>gfix_trn_req</td>
<td>Number of transactions per sweep required</td>
</tr>
<tr>
<td>-901</td>
<td>336330761</td>
<td>gbak_multi_source_dest</td>
<td>Multiple sources or destinations specified</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>335544330</td>
<td>bad_tpb_content</td>
<td>Invalid parameter in transaction parameter block</td>
</tr>
<tr>
<td>-901</td>
<td>336330762</td>
<td>gbak_filename_missing</td>
<td>Requires both input and output filenames</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>336330763</td>
<td>gbak_dup_inout_names</td>
<td>Input and output have the same name. Disallowed</td>
</tr>
<tr>
<td>-901</td>
<td>335740940</td>
<td>gfix_full_req</td>
<td>&quot;full&quot; or &quot;reserve&quot; required</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>336330764</td>
<td>gbak_inv_page_size</td>
<td>Expected page size, encountered &quot;@1&quot;</td>
</tr>
<tr>
<td>-901</td>
<td>335740941</td>
<td>gfix_usename_req</td>
<td>User name required</td>
</tr>
<tr>
<td>-901</td>
<td>336330765</td>
<td>gbak_db_specified</td>
<td>REPLACE specified, but the first file @1 is a database</td>
</tr>
<tr>
<td>-901</td>
<td>335740942</td>
<td>gfix_pass_req</td>
<td>Password required</td>
</tr>
<tr>
<td>-901</td>
<td>336330766</td>
<td>gbak_db_exists</td>
<td>Database @1 already exists. To replace it, use the -REP switch</td>
</tr>
<tr>
<td>-901</td>
<td>335740943</td>
<td>gfix_subs_name</td>
<td>Subsystem name</td>
</tr>
<tr>
<td>-901</td>
<td>336723983</td>
<td>gsec_cant_open_db</td>
<td>Unable to open database</td>
</tr>
<tr>
<td>-901</td>
<td>336330767</td>
<td>gbak_unk_device</td>
<td>Device type not specified</td>
</tr>
<tr>
<td>-901</td>
<td>336723984</td>
<td>gsec_switches_error</td>
<td>Error in switch specifications</td>
</tr>
<tr>
<td>-901</td>
<td>335740945</td>
<td>gfix_sec_req</td>
<td>Number of seconds required</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>336723985</td>
<td>gsec_no_op_spec</td>
<td>No operation specified</td>
</tr>
<tr>
<td>-901</td>
<td>335740946</td>
<td>gfix_nval_req</td>
<td>Numeric value between 0 and 32767 inclusive required</td>
</tr>
<tr>
<td>-901</td>
<td>336723986</td>
<td>gsec_no_usr_name</td>
<td>No user name specified</td>
</tr>
<tr>
<td>-901</td>
<td>335740947</td>
<td>gfix_type_shut</td>
<td>Must specify type of shutdown</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>336723987</td>
<td>gsec_err_add</td>
<td>Add record 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>336723988</td>
<td>gsec_err_modify</td>
<td>Modify record error</td>
</tr>
<tr>
<td>-901</td>
<td>336330772</td>
<td>gbak_blob_info_failed</td>
<td>Gds_$blob_info failed</td>
</tr>
<tr>
<td>-901</td>
<td>335740948</td>
<td>gfix_retry</td>
<td>Please retry, specifying an option</td>
</tr>
<tr>
<td>-901</td>
<td>335544341</td>
<td>infunk</td>
<td>Unknown information item</td>
</tr>
<tr>
<td>-901</td>
<td>336723989</td>
<td>gsec_err_find_mod</td>
<td>Find / modify record error</td>
</tr>
<tr>
<td>-901</td>
<td>336330773</td>
<td>gbak_unk_blob_item</td>
<td>Do not understand BLOB INFO item @1</td>
</tr>
<tr>
<td>-901</td>
<td>335544342</td>
<td>integ_fail</td>
<td>Action cancelled by trigger (@1) to preserve data integrity</td>
</tr>
<tr>
<td>-901</td>
<td>336330774</td>
<td>gbak_get_seg_failed</td>
<td>Gds_$get_segment failed</td>
</tr>
<tr>
<td>-901</td>
<td>336723990</td>
<td>gsec_err_rec_not_found</td>
<td>Record not found for user: @1</td>
</tr>
<tr>
<td>-901</td>
<td>336723991</td>
<td>gsec_err_delete</td>
<td>Delete record error</td>
</tr>
<tr>
<td>-901</td>
<td>336330775</td>
<td>gbak_close_blob_failed</td>
<td>Gds_$close_blob failed</td>
</tr>
<tr>
<td>-901</td>
<td>335740951</td>
<td>gfix_retry_db</td>
<td>Please retry, giving a database name</td>
</tr>
<tr>
<td>-901</td>
<td>336330776</td>
<td>gbak_open_blob_failed</td>
<td>Gds_$open_blob failed</td>
</tr>
<tr>
<td>-901</td>
<td>336723992</td>
<td>gsec_err_find_del</td>
<td>Find / delete record error</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>336330777</td>
<td>gbak_put_blr_gen_id_failed</td>
<td>Failed in put_blr_gen_id</td>
</tr>
<tr>
<td>-901</td>
<td>336330778</td>
<td>gbak_unk_type</td>
<td>Data type @1 not understood</td>
</tr>
<tr>
<td>-901</td>
<td>336330779</td>
<td>gbak_comp_req_failed</td>
<td>Gds_$compile_request failed</td>
</tr>
<tr>
<td>-901</td>
<td>336330780</td>
<td>gbak_start_req_failed</td>
<td>Gds_$start_request failed</td>
</tr>
<tr>
<td>-901</td>
<td>336723996</td>
<td>gsec_err_find_disp</td>
<td>Find / display record error</td>
</tr>
<tr>
<td>-901</td>
<td>336330781</td>
<td>gbak_rec_failed</td>
<td>gds$_receive failed</td>
</tr>
<tr>
<td>-901</td>
<td>336920605</td>
<td>gstat_open_err</td>
<td>Can’t open database file @1</td>
</tr>
<tr>
<td>-901</td>
<td>336723997</td>
<td>gsec_inv_param</td>
<td>Invalid parameter, no switch defined</td>
</tr>
<tr>
<td>-901</td>
<td>335544350</td>
<td>no_finish</td>
<td>Program attempted to exit without finishing database</td>
</tr>
<tr>
<td>-901</td>
<td>336920606</td>
<td>gstat_read_err</td>
<td>Can’t read a database page</td>
</tr>
<tr>
<td>-901</td>
<td>336330782</td>
<td>gbak_rel_req_failed</td>
<td>Gds$_release_request failed</td>
</tr>
<tr>
<td>-901</td>
<td>336723998</td>
<td>gsec_op_specified</td>
<td>Operation already specified</td>
</tr>
<tr>
<td>-901</td>
<td>336920607</td>
<td>gstat_sysmemex</td>
<td>System memory exhausted</td>
</tr>
<tr>
<td>-901</td>
<td>336330783</td>
<td>gbak_db_info_failed</td>
<td>gds$_database_info failed</td>
</tr>
<tr>
<td>-901</td>
<td>336723999</td>
<td>gsec_pw_specified</td>
<td>Password already specified</td>
</tr>
<tr>
<td>SQL-CODE</td>
<td>GDSCODE</td>
<td>Symbol</td>
<td>Message Text</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>---------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>-901</td>
<td>336724000</td>
<td>gsec_uid_specified</td>
<td>Uid already specified</td>
</tr>
<tr>
<td>-901</td>
<td>336330784</td>
<td>gbak_no_db_desc</td>
<td>Expected database description record</td>
</tr>
<tr>
<td>-901</td>
<td>335544353</td>
<td>no_recon</td>
<td>Transaction is not in limbo</td>
</tr>
<tr>
<td>-901</td>
<td>336724001</td>
<td>gsec_gid_specified</td>
<td>Gid already specified</td>
</tr>
<tr>
<td>-901</td>
<td>336330785</td>
<td>gbak_db_create_failed</td>
<td>Failed to create database @1</td>
</tr>
<tr>
<td>-901</td>
<td>336724002</td>
<td>gsec_proj_specified</td>
<td>Project already specified</td>
</tr>
<tr>
<td>-901</td>
<td>336330786</td>
<td>gbak_decomp_len_error</td>
<td>RESTORE: decompression length error</td>
</tr>
<tr>
<td>-901</td>
<td>335544355</td>
<td>no_segstr_close</td>
<td>BLOB was not closed</td>
</tr>
<tr>
<td>-901</td>
<td>336330787</td>
<td>gbak_tbl_missing</td>
<td>Cannot find table @1</td>
</tr>
<tr>
<td>-901</td>
<td>336724003</td>
<td>gsec_org_specified</td>
<td>Organization already specified</td>
</tr>
<tr>
<td>-901</td>
<td>336330788</td>
<td>gbak_blob_col_missing</td>
<td>Cannot find column for BLOB</td>
</tr>
<tr>
<td>-901</td>
<td>336724004</td>
<td>gsec_fname_specified</td>
<td>First name already specified</td>
</tr>
<tr>
<td>-901</td>
<td>335544357</td>
<td>open_trans</td>
<td>Cannot disconnect database with open transactions (@1 active)</td>
</tr>
<tr>
<td>-901</td>
<td>336330789</td>
<td>gbak_create_blob_failed</td>
<td>Gds_$create_blob failed</td>
</tr>
<tr>
<td>-901</td>
<td>336724005</td>
<td>gsec_mname_specified</td>
<td>Middle name already specified</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>336330790</td>
<td>gbak_put_seg_failed</td>
<td>Gds_$put_segment failed</td>
</tr>
<tr>
<td>-901</td>
<td>336724006</td>
<td>gsec_lname_specified</td>
<td>Last name already specified</td>
</tr>
<tr>
<td>-901</td>
<td>336330791</td>
<td>gbak_rec_len_exp</td>
<td>Expected record length</td>
</tr>
<tr>
<td>-901</td>
<td>336724008</td>
<td>gsec_inv_switch</td>
<td>Invalid switch specified</td>
</tr>
<tr>
<td>-901</td>
<td>336330792</td>
<td>gbak_inv_rec_len</td>
<td>Wrong length record, expected @1 encountered @2</td>
</tr>
<tr>
<td>-901</td>
<td>336330793</td>
<td>gbak_exp_data_type</td>
<td>Expected data attribute</td>
</tr>
<tr>
<td>-901</td>
<td>336724009</td>
<td>gsec_amb_switch</td>
<td>Ambiguous switch specified</td>
</tr>
<tr>
<td>-901</td>
<td>336330794</td>
<td>gbak_gen_id_failed</td>
<td>Failed in store_blr_gen_id</td>
</tr>
<tr>
<td>-901</td>
<td>336724010</td>
<td>gsec_no_op_specified</td>
<td>No operation specified for parameters</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>336330795</td>
<td>gbak_unk_rec_type</td>
<td>Do not recognize record type @1</td>
</tr>
<tr>
<td>-901</td>
<td>336724011</td>
<td>gsec_params_not_allowed</td>
<td>No parameters allowed for this operation</td>
</tr>
<tr>
<td>-901</td>
<td>335544364</td>
<td>req_sync</td>
<td>Request synchronization 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>336724012</td>
<td>gsec_incompat_switch</td>
<td>Incompatible switches specified</td>
</tr>
<tr>
<td>-901</td>
<td>336330796</td>
<td>gbak_inv_bkup_ver</td>
<td>Expected backup version 1..8. Found @1</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>336330797</td>
<td>gbak_missing_bkup_desc</td>
<td>Expected backup description record</td>
</tr>
<tr>
<td>-901</td>
<td>336330798</td>
<td>gbak_string_trunc</td>
<td>String truncated</td>
</tr>
<tr>
<td>-901</td>
<td>336330799</td>
<td>gbak_cant_rest_record</td>
<td>warning — record could not be restored</td>
</tr>
<tr>
<td>-901</td>
<td>336330800</td>
<td>gbak_send_failed</td>
<td>Gds_$send failed</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>336330801</td>
<td>gbak_no_tbl_name</td>
<td>No table name for data</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>336330802</td>
<td>gbak_unexp_eof</td>
<td>Unexpected end of file on backup file</td>
</tr>
<tr>
<td>-901</td>
<td>336330803</td>
<td>gbak_db_format_too_old</td>
<td>Database format @1 is too old to restore to</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>336330804</td>
<td>gbak_inv_array_dim</td>
<td>Array dimension for column @1 is invalid</td>
</tr>
<tr>
<td>-901</td>
<td>336330807</td>
<td>gbak_xdr_len_expected</td>
<td>Expected XDR record length</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>-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>335740991</td>
<td>gfix_exceed_max</td>
<td>Internal block exceeds maximum size</td>
</tr>
<tr>
<td>-901</td>
<td>335740992</td>
<td>gfix_corrupt_pool</td>
<td>Corrupt pool</td>
</tr>
<tr>
<td>-901</td>
<td>335740993</td>
<td>gfix_mem_exhausted</td>
<td>Virtual memory exhausted</td>
</tr>
<tr>
<td>-901</td>
<td>336330817</td>
<td>gbak_open_bkup_error</td>
<td>Cannot open backup file @1</td>
</tr>
<tr>
<td>-901</td>
<td>335740994</td>
<td>gbak_bad_pool</td>
<td>Bad pool id.</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>336330818</td>
<td>gbak_open_error</td>
<td>Cannot open status and error output file @1</td>
</tr>
<tr>
<td>-901</td>
<td>335740995</td>
<td>gfix_trn_not_valid</td>
<td>Transaction state @1 not in valid range</td>
</tr>
<tr>
<td>-901</td>
<td>335544392</td>
<td>bdbincon</td>
<td>Internal error</td>
</tr>
<tr>
<td>-901</td>
<td>336724044</td>
<td>gsec_inv_username</td>
<td>Invalid user name (maximum 31 bytes allowed)</td>
</tr>
<tr>
<td>-901</td>
<td>336724045</td>
<td>gsec_inv_pw_length</td>
<td>Warning - maximum 8 significant bytes of password used</td>
</tr>
<tr>
<td>-901</td>
<td>336724046</td>
<td>gsec_db_specified</td>
<td>Database already specified</td>
</tr>
<tr>
<td>-901</td>
<td>336724047</td>
<td>gsec_db_admin_specified</td>
<td>Database administrator name already specified</td>
</tr>
<tr>
<td>-901</td>
<td>336724048</td>
<td>gsec_db_admin_pw_specified</td>
<td>Database administrator password already specified</td>
</tr>
<tr>
<td>-901</td>
<td>336724049</td>
<td>gsec_sql_role_specified</td>
<td>SQL role name already specified</td>
</tr>
<tr>
<td>-901</td>
<td>335741012</td>
<td>gfix_unexp_eoi</td>
<td>Unexpected end of input</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>335741018</td>
<td>gfix_recon_fail</td>
<td>Failed to reconnect to a transaction in database @1</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>335741036</td>
<td>gfix_trn.unknown</td>
<td>Transaction description item unknown</td>
</tr>
<tr>
<td>-901</td>
<td>335741038</td>
<td>gfix_mode_req</td>
<td>&quot;read_only&quot; or &quot;read_write&quot; required</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>335741042</td>
<td>gfix_pzval_req</td>
<td>Positive or zero numeric value required</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>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>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>336330934</td>
<td>gbak_missing_block_fac</td>
<td>Blocking factor parameter missing</td>
</tr>
<tr>
<td>-901</td>
<td>336330935</td>
<td>gbak_inv_block_fac</td>
<td>Expected blocking factor, encountered &quot;@1&quot;</td>
</tr>
<tr>
<td>-901</td>
<td>336330936</td>
<td>gbak_block_fac_specified</td>
<td>A blocking factor may not be used in conjunction with device CT</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>336330940</td>
<td>gbak_missing_username</td>
<td>User name parameter missing</td>
</tr>
<tr>
<td>-901</td>
<td>336330941</td>
<td>gbak_missing_password</td>
<td>Password parameter missing</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>33554510</td>
<td>lock_timeout</td>
<td>Lock time-out on wait transaction</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>336330952</td>
<td>gbak_missing_skipped_bytes</td>
<td>missing parameter for the number of bytes to be skipped</td>
</tr>
<tr>
<td>-901</td>
<td>336330953</td>
<td>gbak_inv_skipped_bytes</td>
<td>Expected number of bytes to be skipped, encountered &quot;@1&quot;</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>336330965</td>
<td>gbak_err_restore_charset</td>
<td>Character set</td>
</tr>
<tr>
<td>-901</td>
<td>336330967</td>
<td>gbak_err_restore_collation</td>
<td>Collation</td>
</tr>
<tr>
<td>-901</td>
<td>336330972</td>
<td>gbak_read_error</td>
<td>Unexpected I/O error while reading from backup file</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>336330973</td>
<td>gbak_write_error</td>
<td>Unexpected I/O error while writing to backup file</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>336330985</td>
<td>gbak_db_in_use</td>
<td>Could not drop database @1 (database might be in use)</td>
</tr>
<tr>
<td>-901</td>
<td>336330990</td>
<td>gbak_sysmemex</td>
<td>System memory exhausted</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>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>336331002</td>
<td>gbak_restore_role_failed</td>
<td>SQL role</td>
</tr>
<tr>
<td>-901</td>
<td>336331005</td>
<td>gbak_role_op_missing</td>
<td>SQL role parameter missing</td>
</tr>
<tr>
<td>-901</td>
<td>336331010</td>
<td>gbak_page_buffers_missing</td>
<td>Page buffers parameter missing</td>
</tr>
<tr>
<td>-901</td>
<td>336331011</td>
<td>gbak_page_buffers_wrong_param</td>
<td>Expected page buffers, encountered &quot;@1&quot;</td>
</tr>
<tr>
<td>-901</td>
<td>336331012</td>
<td>gbak_page_buffers_restore</td>
<td>Page buffers is allowed only on restore or create</td>
</tr>
<tr>
<td>-901</td>
<td>336331014</td>
<td>gbak_inv_size</td>
<td>Size specification either missing or incorrect for file @1</td>
</tr>
<tr>
<td>-901</td>
<td>336331015</td>
<td>gbak_file_outof_sequence</td>
<td>File @1 out of sequence</td>
</tr>
<tr>
<td>-901</td>
<td>336331016</td>
<td>gbak_join_file_missing</td>
<td>Can't join - one of the files missing</td>
</tr>
<tr>
<td>-901</td>
<td>336331017</td>
<td>gbak_stdin_not_suppted</td>
<td>Standard input is not supported when using join operation</td>
</tr>
<tr>
<td>-901</td>
<td>336331018</td>
<td>gbak_stdout_not_supptd</td>
<td>Standard output is not supported when using split operation</td>
</tr>
<tr>
<td>-901</td>
<td>336331019</td>
<td>gbak_bkup_corrupt</td>
<td>Backup file @1 might be corrupt</td>
</tr>
<tr>
<td>-901</td>
<td>336331020</td>
<td>gbak_unk_db_file_spec</td>
<td>Database file specification missing</td>
</tr>
<tr>
<td>-901</td>
<td>336331021</td>
<td>gbak_hdr_write_failed</td>
<td>Can't write a header record to file @1</td>
</tr>
<tr>
<td>-901</td>
<td>336331022</td>
<td>gbak_disk_space_ex</td>
<td>Free disk space exhausted</td>
</tr>
<tr>
<td>-901</td>
<td>336331023</td>
<td>gbak_size_lt_min</td>
<td>File size given (@1) is less than minimum allowed (@2)</td>
</tr>
<tr>
<td>-901</td>
<td>336331025</td>
<td>gbak_svc_name_missing</td>
<td>Service name parameter missing</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>336331026</td>
<td>gbak_not_ownr</td>
<td>Cannot restore over current database, must be SYSDBA or owner of the existing database</td>
</tr>
<tr>
<td>-901</td>
<td>336331031</td>
<td>gbak_mode_req</td>
<td>&quot;read_only&quot; or &quot;read_write&quot; required</td>
</tr>
<tr>
<td>-901</td>
<td>336331033</td>
<td>gbak_just_data</td>
<td>Just data ignore all constraints etc.</td>
</tr>
<tr>
<td>-901</td>
<td>336331034</td>
<td>gbak_data_only</td>
<td>Restoring data only ignoring foreign key, unique, not null &amp; other constraints</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>
</tbody>
</table>

Table 188. SQLCODE and GDSCODE Error Codes and Message Texts (4)
<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>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>335544731</td>
<td>tra.must_sweep</td>
<td>[no associated message]</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>-901</td>
<td>335544741</td>
<td>lost_db_connection</td>
<td>Connection lost to database</td>
</tr>
<tr>
<td>-901</td>
<td>335544742</td>
<td>no_write_user_priv</td>
<td>User cannot write to RDB$USER_PRIVILEGES</td>
</tr>
<tr>
<td>-901</td>
<td>335544767</td>
<td>blob.filter_exception</td>
<td>A fatal exception occurred during the execution of a blob filter</td>
</tr>
<tr>
<td>-901</td>
<td>335544768</td>
<td>exception_access_violation</td>
<td>Access violation. The code attempted to access a virtual address without privilege to do so</td>
</tr>
<tr>
<td>-901</td>
<td>335544769</td>
<td>exception_datatype_missalignment</td>
<td>Datatype misalignment. The attempted to read or write a value that was not stored on a memory boundary</td>
</tr>
<tr>
<td>-901</td>
<td>335544770</td>
<td>exception_array_bounds_exceeded</td>
<td>Array bounds exceeded. The code attempted to access an array element that is out of bounds.</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>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>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_sigsbus</td>
<td>Bus Error. The Code caused a system bus error</td>
</tr>
<tr>
<td>-901</td>
<td>335544785</td>
<td>exception_sigsfpe</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>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>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>-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>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>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>336397211</td>
<td>dsqlo_to_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>dsqlo_unsupp_feature_dialect</td>
<td>Feature is not supported in dialect @1</td>
</tr>
<tr>
<td>-902</td>
<td>335544333</td>
<td>bug_check</td>
<td>Internal gds software consistency check (@1)</td>
</tr>
<tr>
<td>-902</td>
<td>335544335</td>
<td>db_crupt</td>
<td>Database file appears corrupt (@1)</td>
</tr>
<tr>
<td>-902</td>
<td>335544344</td>
<td>io_error</td>
<td>I/O error for file &quot;@2&quot;</td>
</tr>
<tr>
<td>-902</td>
<td>335544346</td>
<td>metadata_crupt</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>-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>dirtypage</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>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>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>sqlerr</td>
<td>SQL error code = @1</td>
</tr>
<tr>
<td>-902</td>
<td>335544448</td>
<td>bad_sec_info</td>
<td>[no associated message]</td>
</tr>
<tr>
<td>-902</td>
<td>335544449</td>
<td>invalid_sec_info</td>
<td>[no associated message]</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>-902</td>
<td>33554506</td>
<td>shutinprog</td>
<td>Database shutdown in progress</td>
</tr>
<tr>
<td>-902</td>
<td>33554528</td>
<td>shutdown</td>
<td>Database @1 shutdown</td>
</tr>
<tr>
<td>-902</td>
<td>33554557</td>
<td>shutfail</td>
<td>Database shutdown unsuccessful</td>
</tr>
<tr>
<td>-902</td>
<td>33554569</td>
<td>dsql_error</td>
<td>Dynamic SQL Error</td>
</tr>
<tr>
<td>-902</td>
<td>33554653</td>
<td>psw_attach</td>
<td>Cannot attach to password database</td>
</tr>
<tr>
<td>-902</td>
<td>33554654</td>
<td>psw_start_trans</td>
<td>Cannot start transaction for password database</td>
</tr>
<tr>
<td>-902</td>
<td>33554717</td>
<td>err_stack_limit</td>
<td>Stack size insufficient to execute current request</td>
</tr>
<tr>
<td>-902</td>
<td>33554721</td>
<td>network_error</td>
<td>Unable to complete network request to host &quot;@1&quot;</td>
</tr>
<tr>
<td>-902</td>
<td>33554722</td>
<td>net_connect_err</td>
<td>Failed to establish a connection</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>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>
<tr>
<td>-902</td>
<td>335544726</td>
<td>net_read_err</td>
<td>Error reading data from the connection</td>
</tr>
<tr>
<td>-902</td>
<td>335544727</td>
<td>net_write_err</td>
<td>Error writing data to the connection</td>
</tr>
<tr>
<td>-902</td>
<td>335544732</td>
<td>unsupported_network_drive</td>
<td>Access to databases on file servers is not supported</td>
</tr>
<tr>
<td>-902</td>
<td>335544733</td>
<td>io_create_err</td>
<td>Error while trying to create file</td>
</tr>
<tr>
<td>-902</td>
<td>335544734</td>
<td>io_open_err</td>
<td>Error while trying to open file</td>
</tr>
<tr>
<td>-902</td>
<td>335544735</td>
<td>io_close_err</td>
<td>Error while trying to close file</td>
</tr>
<tr>
<td>-902</td>
<td>335544736</td>
<td>io_read_err</td>
<td>Error while trying to read from file</td>
</tr>
<tr>
<td>-902</td>
<td>335544737</td>
<td>io_write_err</td>
<td>Error while trying to write to file</td>
</tr>
<tr>
<td>-902</td>
<td>335544738</td>
<td>io_delete_err</td>
<td>Error while trying to delete file</td>
</tr>
<tr>
<td>-902</td>
<td>335544739</td>
<td>io_access_err</td>
<td>Error while trying to access file</td>
</tr>
<tr>
<td>-902</td>
<td>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</td>
<td>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</td>
<td>335544795</td>
<td>unexp_spb_form</td>
<td>Unexpected item in service parameter block, expected @1</td>
</tr>
<tr>
<td>-902</td>
<td>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</td>
<td>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</td>
<td>335544831</td>
<td>conf_access_denied</td>
<td>Access to @1 &quot;@2&quot; is denied by server administrator</td>
</tr>
<tr>
<td>-902</td>
<td>335544834</td>
<td>cursor_not_open</td>
<td>Cursor is not open</td>
</tr>
<tr>
<td>-902</td>
<td>335544841</td>
<td>cursor_already_open</td>
<td>Cursor is already open</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>335544856</td>
<td>att_shutdown</td>
<td>Connection shutdown</td>
</tr>
<tr>
<td>-902</td>
<td>335544882</td>
<td>long_login</td>
<td>Login name too long (@1 characters, maximum allowed @2)</td>
</tr>
<tr>
<td>-904</td>
<td>335544324</td>
<td>bad_db_handle</td>
<td>Invalid database handle (no active connection)</td>
</tr>
<tr>
<td>-904</td>
<td>335544375</td>
<td>unavailable</td>
<td>Unavailable database</td>
</tr>
<tr>
<td>-904</td>
<td>335544381</td>
<td>imp_exc</td>
<td>Implementation limit exceeded</td>
</tr>
<tr>
<td>-904</td>
<td>335544386</td>
<td>nopoolids</td>
<td>Too many requests</td>
</tr>
<tr>
<td>-904</td>
<td>335544389</td>
<td>bufexh</td>
<td>Buffer exhausted</td>
</tr>
<tr>
<td>-904</td>
<td>335544391</td>
<td>bufinuse</td>
<td>Buffer in use</td>
</tr>
<tr>
<td>-904</td>
<td>335544393</td>
<td>reqinuse</td>
<td>Request in use</td>
</tr>
<tr>
<td>-904</td>
<td>335544424</td>
<td>no_lock_mgr</td>
<td>No lock manager available</td>
</tr>
<tr>
<td>-904</td>
<td>335544430</td>
<td>virmemexh</td>
<td>Unable to allocate memory from operating system</td>
</tr>
<tr>
<td>-904</td>
<td>335544451</td>
<td>update_conflict</td>
<td>Update conflicts with concurrent update</td>
</tr>
<tr>
<td>-904</td>
<td>335544453</td>
<td>obj_in_use</td>
<td>Object @1 is in use</td>
</tr>
<tr>
<td>-904</td>
<td>335544455</td>
<td>shadow_accessed</td>
<td>Cannot attach active shadow file</td>
</tr>
<tr>
<td>-904</td>
<td>335544460</td>
<td>shadow_missing</td>
<td>A file in manual shadow @1 is unavailable</td>
</tr>
<tr>
<td>-904</td>
<td>335544661</td>
<td>index_root_page_full</td>
<td>Cannot add index, index root page is full</td>
</tr>
<tr>
<td>-904</td>
<td>335544676</td>
<td>sort_mem_err</td>
<td>Sort error: not enough memory</td>
</tr>
<tr>
<td>-904</td>
<td>335544683</td>
<td>req_depth_exceeded</td>
<td>Request depth exceeded. (Recursive definition?)</td>
</tr>
<tr>
<td>-904</td>
<td>335544758</td>
<td>sort_rec_size_err</td>
<td>Sort record size of @1 bytes is too big ???????</td>
</tr>
<tr>
<td>-904</td>
<td>335544761</td>
<td>too_many_handles</td>
<td>Too many open handles to database</td>
</tr>
<tr>
<td>-904</td>
<td>335544792</td>
<td>service_att_err</td>
<td>Cannot attach to services manager</td>
</tr>
<tr>
<td>-904</td>
<td>335544799</td>
<td>svc_name_missing</td>
<td>The service name was not specified</td>
</tr>
<tr>
<td>-904</td>
<td>335544813</td>
<td>optimizer_between_err</td>
<td>Unsupported field type specified in BETWEEN predicate</td>
</tr>
<tr>
<td>-904</td>
<td>335544827</td>
<td>exec_sql_invalid_arg</td>
<td>Invalid argument in EXECUTE STATEMENT-cannot convert to string</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>-904</td>
<td>335544828</td>
<td>exec_sql_invalid_req</td>
<td>Wrong request type in EXECUTE STATEMENT '@1'</td>
</tr>
<tr>
<td>-904</td>
<td>335544829</td>
<td>exec_sql_invalid_var</td>
<td>Variable type (position @1) in EXECUTE STATEMENT '@2' INTO does not match returned column type</td>
</tr>
<tr>
<td>-904</td>
<td>335544830</td>
<td>exec_sql_max_call_exceeded</td>
<td>Too many recursion levels of EXECUTE STATEMENT</td>
</tr>
<tr>
<td>-904</td>
<td>335544832</td>
<td>wrong_backup_state</td>
<td>Cannot change difference file name while database is in backup mode</td>
</tr>
<tr>
<td>-904</td>
<td>335544852</td>
<td>partner_idx_incompat_type</td>
<td>Partner index segment no @1 has incompatible data type</td>
</tr>
<tr>
<td>-904</td>
<td>335544857</td>
<td>blobtoobig</td>
<td>Maximum BLOB size exceeded</td>
</tr>
<tr>
<td>-904</td>
<td>335544862</td>
<td>record_lock_not_supp</td>
<td>Stream does not support record locking</td>
</tr>
<tr>
<td>-904</td>
<td>335544863</td>
<td>partner_idx_not_found</td>
<td>Cannot create foreign key constraint @1. Partner index does not exist or is inactive</td>
</tr>
<tr>
<td>-904</td>
<td>335544864</td>
<td>tra_num_exc</td>
<td>Transactions count exceeded. Perform backup and restore to make database operable again</td>
</tr>
<tr>
<td>-904</td>
<td>335544865</td>
<td>field_disappeared</td>
<td>Column has been unexpectedly deleted</td>
</tr>
<tr>
<td>-904</td>
<td>335544878</td>
<td>concurrent_transaction</td>
<td>Concurrent transaction number is @1</td>
</tr>
<tr>
<td>-906</td>
<td>335544744</td>
<td>max_att_exceeded</td>
<td>Maximum user count exceeded. Contact your database administrator</td>
</tr>
<tr>
<td>-909</td>
<td>335544667</td>
<td>drdb_completed_with_errs</td>
<td>Drop database completed with errors</td>
</tr>
<tr>
<td>-911</td>
<td>335544459</td>
<td>rec_in_limbo</td>
<td>Record from transaction @1 is stuck in limbo</td>
</tr>
<tr>
<td>-913</td>
<td>335544336</td>
<td>deadlock</td>
<td>Deadlock</td>
</tr>
<tr>
<td>-922</td>
<td>335544323</td>
<td>bad_db_format</td>
<td>File @1 is not a valid database</td>
</tr>
<tr>
<td>-923</td>
<td>335544421</td>
<td>connect_reject</td>
<td>Connection rejected by remote interface</td>
</tr>
<tr>
<td>-923</td>
<td>335544461</td>
<td>cant_validate</td>
<td>Secondary server attachments cannot validate databases</td>
</tr>
<tr>
<td>-923</td>
<td>335544464</td>
<td>cant_start_logging</td>
<td>Secondary server attachments cannot start logging</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>-924</td>
<td>335544325</td>
<td>bad_dpb_content</td>
<td>Bad parameters on attach or create database</td>
</tr>
<tr>
<td>-924</td>
<td>335544441</td>
<td>bad_detach</td>
<td>Database detach completed with errors</td>
</tr>
<tr>
<td>-924</td>
<td>335544648</td>
<td>conn_lost</td>
<td>Connection lost to pipe server</td>
</tr>
<tr>
<td>-926</td>
<td>335544447</td>
<td>no_rollback</td>
<td>No rollback performed</td>
</tr>
<tr>
<td>-999</td>
<td>335544689</td>
<td>ib_error</td>
<td>Firebird 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 2.5:

- ADD
- ADMIN
- ALL
- ALTER
- AND
- ANY
- AS
- AT
- AVG
- BEGIN
- BETWEEN
- BIGINT
- BIT_LENGTH
- BLOB
- BOTH
- BY
- CASE
- CAST
- CHAR
- CHAR_LENGTH
- CHARACTER
- CHARACTER_LENGTH
- CHECK
- CLOSE
- COLLABORATOR
- COLUMN
- COMMIT
- CONNECT
- CONSTRAINT
- COUNT
- CREATE
- CROSS
- CURRENT
- CURRENT_CONNECTION
- CURRENT_DATE
- CURRENT_ROLE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_TRANSACTION
- DATE
- DEC
- DECIMAL
- DECLARE
- DEFAULT
- DELETE
- DELETING
- DISCONNECT
- DISTINCT
- DOUBLE
- DROP
- ELSE
- END
- ESCAPE
- EXECUTE
- EXISTS
- EXTERNAL
- EXTRACT
- FETCH
- FILTER
- FLOAT
- FOR
- FOREIGN
- FROM
- FULL
- FUNCTION
- GDSCODE
- GLOBAL
- GRANT
- GROUP
- HAVING
- HOUR
- IN
- INDEX
- INNER
- INSENSITIVE
- INSERT
- INSERTING
- INT
The following terms have a special meaning in Firebird 2.5 DSQL. Some of them are also reserved words, others are not.

<!> ^< ^=
^> , :=
!= !> ()
) < <=
<table>
<thead>
<tr>
<th>Keywords</th>
<th>Reserved Words and Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td></td>
</tr>
<tr>
<td><code>~=</code></td>
<td></td>
</tr>
<tr>
<td><code>ACOS</code></td>
<td></td>
</tr>
<tr>
<td><code>ADD</code></td>
<td></td>
</tr>
<tr>
<td><code>ACTION</code></td>
<td></td>
</tr>
<tr>
<td><code>ADMIN</code></td>
<td></td>
</tr>
<tr>
<td><code>ALTER</code></td>
<td></td>
</tr>
<tr>
<td><code>ANY</code></td>
<td></td>
</tr>
<tr>
<td><code>ASC</code></td>
<td></td>
</tr>
<tr>
<td><code>ASCENDING</code></td>
<td></td>
</tr>
<tr>
<td><code>ASIN</code></td>
<td></td>
</tr>
<tr>
<td><code>AT</code></td>
<td></td>
</tr>
<tr>
<td><code>ATAN</code></td>
<td></td>
</tr>
<tr>
<td><code>ATAN2</code></td>
<td></td>
</tr>
<tr>
<td><code>AVG</code></td>
<td></td>
</tr>
<tr>
<td><code>BEGIN</code></td>
<td></td>
</tr>
<tr>
<td><code>BETWEEN</code></td>
<td></td>
</tr>
<tr>
<td><code>BIN_AND</code></td>
<td></td>
</tr>
<tr>
<td><code>BIN_OR</code></td>
<td></td>
</tr>
<tr>
<td><code>BIN_SHL</code></td>
<td></td>
</tr>
<tr>
<td><code>BIT_LENGTH</code></td>
<td></td>
</tr>
<tr>
<td><code>BLOB</code></td>
<td></td>
</tr>
<tr>
<td><code>BLOCK</code></td>
<td></td>
</tr>
<tr>
<td><code>BOTH</code></td>
<td></td>
</tr>
<tr>
<td><code>CALLER</code></td>
<td></td>
</tr>
<tr>
<td><code>CAST</code></td>
<td></td>
</tr>
<tr>
<td><code>CHAR</code></td>
<td></td>
</tr>
<tr>
<td><code>CHARACTER</code></td>
<td></td>
</tr>
<tr>
<td><code>CHECK</code></td>
<td></td>
</tr>
<tr>
<td><code>CLOSE</code></td>
<td></td>
</tr>
<tr>
<td><code>COALESCE</code></td>
<td></td>
</tr>
<tr>
<td><code>COLLATE</code></td>
<td></td>
</tr>
<tr>
<td><code>COMMIT</code></td>
<td></td>
</tr>
<tr>
<td><code>CONNECT</code></td>
<td></td>
</tr>
<tr>
<td><code>CONSTRAINT</code></td>
<td></td>
</tr>
<tr>
<td><code>DATEADD</code></td>
<td></td>
</tr>
<tr>
<td><code>DATEDIFF</code></td>
<td></td>
</tr>
<tr>
<td><code>DEC</code></td>
<td></td>
</tr>
<tr>
<td><code>DECLARE</code></td>
<td></td>
</tr>
<tr>
<td><code>DECODE</code></td>
<td></td>
</tr>
<tr>
<td><code>DELETE</code></td>
<td></td>
</tr>
<tr>
<td><code>DELETING</code></td>
<td></td>
</tr>
<tr>
<td><code>DESC</code></td>
<td></td>
</tr>
<tr>
<td><code>DESCENDING</code></td>
<td></td>
</tr>
<tr>
<td><code>DESCRIPTOR</code></td>
<td></td>
</tr>
<tr>
<td><code>DISTINCT</code></td>
<td></td>
</tr>
<tr>
<td><code>DO</code></td>
<td></td>
</tr>
<tr>
<td><code>DOMAIN</code></td>
<td></td>
</tr>
<tr>
<td><code>DOUBLE</code></td>
<td></td>
</tr>
<tr>
<td><code>DROP</code></td>
<td></td>
</tr>
<tr>
<td><code>ELSE</code></td>
<td></td>
</tr>
<tr>
<td><code>END</code></td>
<td></td>
</tr>
<tr>
<td>Reserved Word</td>
<td>Reserved Word</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>ENTRY_POINT</td>
<td>ESCAPE</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>EXISTS</td>
</tr>
<tr>
<td>EXP</td>
<td>EXTERNAL</td>
</tr>
<tr>
<td>FETCH</td>
<td>FILE</td>
</tr>
<tr>
<td>FIRST</td>
<td>FIRSTNAME</td>
</tr>
<tr>
<td>FLOOR</td>
<td>FOR</td>
</tr>
<tr>
<td>FREE_IT</td>
<td>FROM</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>GDSOURCE</td>
</tr>
<tr>
<td>GEN_UUID</td>
<td>GENERATED</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>GRANT</td>
</tr>
<tr>
<td>GROUP</td>
<td>HASH</td>
</tr>
<tr>
<td>HOUR</td>
<td>IF</td>
</tr>
<tr>
<td>IIF</td>
<td>IN</td>
</tr>
<tr>
<td>INDEX</td>
<td>INNER</td>
</tr>
<tr>
<td>INSENSITIVE</td>
<td>INSERT</td>
</tr>
<tr>
<td>INT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>IS</td>
<td>ISOLATION</td>
</tr>
<tr>
<td>KEY</td>
<td>LAST</td>
</tr>
<tr>
<td>LEADING</td>
<td>LEAVE</td>
</tr>
<tr>
<td>LENGTH</td>
<td>LEVEL</td>
</tr>
<tr>
<td>LIMBO</td>
<td>LIST</td>
</tr>
<tr>
<td>LOCK</td>
<td>LOG</td>
</tr>
<tr>
<td>LONG</td>
<td>LOWER</td>
</tr>
<tr>
<td>MANUAL</td>
<td>MAPPING</td>
</tr>
<tr>
<td>MATCHING</td>
<td>MAX</td>
</tr>
<tr>
<td>MAXVALUE</td>
<td>MERGE</td>
</tr>
<tr>
<td>MILLISECOND</td>
<td>MIN</td>
</tr>
<tr>
<td>MINVALUE</td>
<td>MOD</td>
</tr>
<tr>
<td>MONTH</td>
<td>NAMES</td>
</tr>
<tr>
<td>NATURAL</td>
<td>NCHAR</td>
</tr>
<tr>
<td>NO</td>
<td>NOT</td>
</tr>
<tr>
<td>NULLIF</td>
<td>NULLS</td>
</tr>
<tr>
<td>OCTET_LENGTH</td>
<td>OF</td>
</tr>
<tr>
<td>ONLY</td>
<td>OPEN</td>
</tr>
<tr>
<td>OR</td>
<td>ORDER</td>
</tr>
<tr>
<td>OUTER</td>
<td>OUTPUT_TYPE</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>OVERLAY</td>
</tr>
<tr>
<td>PAGE</td>
<td>PAGE_SIZE</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>PASSWORD</td>
</tr>
<tr>
<td>Reserved Word</td>
<td>SQL Code</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
</tr>
<tr>
<td>Appendix C: Reserved Words and Keywords</td>
<td>486</td>
</tr>
<tr>
<td>WRITE</td>
<td>YEAR</td>
</tr>
<tr>
<td>-------</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$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$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

RDB$FILTERS
   Information about BLOB filters

RDB$FORMATS
   Information about changes in the formats of tables

RDB$FUNCTIONS
   Information about external functions
Appendix D: System Tables

RDB$FUNCTION_ARGUMENTS
Attributes of the parameters of external functions

RDB$GENERATORS
Information about generators (sequences)

RDB$INDEX_SEGMENTS
Segments and index positions

RDB$INDICES
Definitions of all indexes in the database (system- or user-defined)

RDB$LOG_FILES
Not used in the current version

RDB$PAGES
Information about database pages

RDB$PROCEDURE_PARAMETERS
Parameters of stored procedures

RDB$PROCEDURES
Definitions of stored procedures

RDB$REF_CONSTRAINTS
Definitions of referential constraints (foreign keys)

RDB$RELATION_CONSTRAINTS
Definitions of all table-level constraints

RDB$RELATION_FIELDS
Top-level definitions of table columns

RDB$RELATIONS
Headers of tables and views

RDB$ROLES
Role definitions

RDB$SECURITY_CLASSES
Access control lists

RDB$TRANSACTIONS
State of multi-database transactions

RDB$TRIGGER_MESSAGES
Trigger messages
RDB$TRIGGERS
Trigger definitions

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$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>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Could store text description of the character set</td>
</tr>
</tbody>
</table>
### Column Name | Data Type | Description
---|---|---
RDB$FUNCTION_NAME | CHAR(31) | For a user-defined character set that is accessed via an external function, the name of the external function.
RDB$BYTES_PER_CHARACTER | SMALLINT | The maximum number of bytes representing one character.

### 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).

| Column Name | Data Type | Description
---|---|---
RDB$CONSTRAINT_NAME | CHAR(31) | Constraint name, defined by the user or automatically generated by the system.
RDB$TRIGGER_NAME | CHAR(31) | 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.

### RDB$COLLATIONS

RDB$COLLATIONS stores collation sequences for all character sets.

| Column Name | Data Type | Description
---|---|---
RDB$COLLATION_NAME | CHAR(31) | Collation sequence name.
RDB$COLLATION_ID | SMALLINT | Collation sequence identifier. Together with the character set identifier, it is a unique collation sequence identifier.
RDB$CHARACTER_SET_ID | SMALLINT | Character set identifier. Together with the collection sequence identifier, it is a unique identifier.
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$COLLATION_ATTRIBUTES</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>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Flag: the value of 0 means it is user-defined; the value of 1 means it is system-defined</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Could store text description of the collation sequence</td>
</tr>
<tr>
<td>RDB$FUNCTION_NAME</td>
<td>CHAR(31)</td>
<td>Not currently used</td>
</tr>
<tr>
<td>RDB$BASE_COLLATION_NAME</td>
<td>CHAR(31)</td>
<td>The name of the base collation sequence for this collation sequence</td>
</tr>
<tr>
<td>RDB$SPECIFIC_ATTRIBUTES</td>
<td>BLOB TEXT</td>
<td>Describes specific attributes</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>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Database comment text</td>
</tr>
<tr>
<td>RDB$RELATION_ID</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>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$CHARACTER_SET_NAME</td>
<td>CHAR(31)</td>
<td>The name of the default character set for the database set in the DEFAULT CHARACTER_SET clause when the database is created. NULL for character set NONE.</td>
</tr>
</tbody>
</table>
RDB$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>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>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17 - collation</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>
</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$FIELD_NAME</td>
<td>CHAR(31)</td>
<td>The unique name of the domain created by the user or of the domain automatically built for the table column by the system. System-created domain names start with the “RDB$” 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 the check of the CHECK value in the domain.</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. 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>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></td>
<td></td>
<td>Codes for DECIMAL and NUMERIC are the same as for the integer types used to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>store them</td>
</tr>
<tr>
<td>RDB$FIELD_SUB_TYPE</td>
<td>SMALLINT</td>
<td>Specifies the subtype for the BLOB data type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - untyped</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - BLR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - access control list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 - reserved for future use</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 - encoded table metadata description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 - for storing the details of a cross-database transaction that ends</td>
</tr>
<tr>
<td></td>
<td></td>
<td>abnormally</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies for the CHAR data type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - untyped data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - fixed binary data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifies the particular data type for the integer data types (SMALLINT,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER, BIGINT) and for fixed-point numbers (NUMERIC, DECIMAL):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 or NULL - the data type matches the value in the RDB$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>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</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$EDIT_STRING</td>
<td>VARCHAR(127)</td>
<td>Not used</td>
</tr>
<tr>
<td>RDB$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>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>
</tbody>
</table>
### Column Name | Data Type | Description
--- | --- | ---
**RDB$CHARACTER_LENGTH** | SMALLINT | The length of CHAR or VARCHAR columns in characters (not in bytes)
**RDB$COLLATION_ID** | SMALLINT | 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
**RDB$CHARACTER_SET_ID** | SMALLINT | The identifier of the character set for a character column, BLOB, TEXT column or domain
**RDB$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$FIELD_DIMENSIONS

RDB$FIELD_DIMENSIONS stores the dimensions of array columns.

| Column Name     | Data Type | Description |
--- | --- | ---
**RDB$FIELD_NAME** | CHAR(31) | The name of the array column. It must be present in the RDB$FIELD_NAME field of the RDB$FIELDS table
**RDB$DIMENSION** | SMALLINT | Identifies one dimension in the array column. The numbering of dimensions starts with 0
**RDB$LOWER_BOUND** | INTEGER | The lower bound of this dimension
**RDB$UPPER_BOUND** | INTEGER | The upper bound of this dimension

### RDB$FILES

RDB$FILES stores information about secondary files and shadow files.

| Column Name     | Data Type | Description |
--- | --- | ---
**RDB$FILE_NAME** | VARCHAR(255) | The full path to the file and the name of either
- the database secondary file in a multi-file database, or
- the shadow file
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$FILE_SEQUENCE</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>RDB$FILE_START</td>
<td>INTEGER</td>
<td>The initial page number in the secondary file or shadow file</td>
</tr>
<tr>
<td>RDB$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(31)</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: 0 - user-defined 1 or greater - internally defined</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
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 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 time the format record was created</td>
</tr>
</tbody>
</table>

**RDB$FUNCTIONS**

RDB$FUNCTIONS stores the information needed by the engine about external functions (user-defined functions, UDFs).

In future major releases (Firebird 3.0+) RDB$FUNCTIONS will also store the information about stored functions: user-defined functions written in PSQL.

<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$MODULE_NAME</td>
<td>VARCHAR(255)</td>
<td>The name of the dynamic library or shared object where the code of the external function is located</td>
</tr>
<tr>
<td>RDB$ENTRYPOINT</td>
<td>CHAR(31)</td>
<td>The exported name of the external function in the function library. Note, this is often not the same as RDB$FUNCTION_NAME, which is the identifier with which the external function is declared to the database</td>
</tr>
<tr>
<td>RDB$RETURN_ARGUMENT</td>
<td>SMALLINT</td>
<td>The position number of the returned argument in the list of parameters corresponding to input arguments</td>
</tr>
</tbody>
</table>
### RDB$SYSTEM_FLAG

**Column Name**: RDB$SYSTEM_FLAG  
**Data Type**: SMALLINT  
**Description**: Flag indicating whether the filter is user-defined or internally defined: 
- 0 - user-defined  
- 1 - internally defined

---

### RDB$FUNCTION_ARGUMENTS

**Column Name**: RDB$FUNCTION_ARGUMENTS  
**Description**: 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$FUNCTION_NAME</td>
<td>CHAR(31)</td>
<td>The unique name (declared identifier) of the external function</td>
</tr>
<tr>
<td>RDB$ARGUMENT_POSITION</td>
<td>SMALLINT</td>
<td>The position of the argument in the list of arguments</td>
</tr>
<tr>
<td>RDB$MECHANISM</td>
<td>SMALLINT</td>
<td>Flag: how this argument is passed:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 0 - by value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 1 - by reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 2 - by descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 3 - by BLOB descriptor</td>
</tr>
<tr>
<td>RDB$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>- 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>
</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$FIELD_LENGTH</td>
<td>SMALLINT</td>
<td>Argument length in bytes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SMALLINT = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER = 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DATE = 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIME = 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BIGINT = 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DOUBLE PRECISION = 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIMESTAMP = 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BLOB_ID = 8</td>
</tr>
<tr>
<td>RDB$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$CHARACTER_SET_ID</td>
<td>SMALLINT</td>
<td>The identifier of the character set for a character argument</td>
</tr>
<tr>
<td>RDB$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>
</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$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - user-defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 or greater - system-defined</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Could store comments related to 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$INDEX_NAME</td>
<td>CHAR(31)</td>
<td>The unique name of the index specified by the user or automatically generated by the system</td>
</tr>
<tr>
<td>RDB$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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - not unique</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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$INDEX_INACTIVE</td>
<td>SMALLINT</td>
<td>Indicates whether the index is currently active:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - active</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - inactive</td>
</tr>
<tr>
<td>RDB$INDEX_TYPE</td>
<td>SMALLINT</td>
<td>Distinguishes between an expression index (1) and a regular index (0 or null). Not used in databases created before Firebird 2.0; hence, regular indexes in upgraded databases are more more likely to store null in this column</td>
</tr>
<tr>
<td>RDB$FOREIGN_KEY</td>
<td>CHAR(31)</td>
<td>The name of the associated Foreign Key constraint, if any</td>
</tr>
<tr>
<td>RDB$SYSTEM_FLAG</td>
<td>SMALLINT</td>
<td>Indicates whether the index is system-defined or user-defined:</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>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>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$STATISTICS</td>
<td>DOUBLE PRECISION</td>
<td>Stores the last known selectivity of the entire index, calculated by execution of a SET STATISTICS statement over the index. It is also recalculated whenever the database is first opened by the server. The selectivity of each separate segment of the index is stored in RDB$INDEX_SEGMENTS.</td>
</tr>
</tbody>
</table>

**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$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>
</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$PROCEDURE_INPUTS</td>
<td>SMALLINT</td>
<td>Indicates the number of input parameters. NULL if there are none</td>
</tr>
<tr>
<td>RDB$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>
</tbody>
</table>
### Appendix D: System Tables

#### RDB$PROCEDURE_TYPE

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

#### RDB$VALID_BLR

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

#### RDB$DEBUG_INFO

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB$DEBUG_INFO</td>
<td>BLOB</td>
<td>Contains debugging information about variables used in the stored procedure</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>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</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$PARAMETER_MECHANISM</td>
<td>SMALLINT</td>
<td>Flag: indicates how this parameter is passed:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - by value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - by reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - by descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - by BLOB descriptor</td>
</tr>
<tr>
<td>RDB$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>
</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$MATCH_OPTION</td>
<td>CHAR(7)</td>
<td>Not used. The current value is FULL in all cases</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$UPDATE_RULE</td>
<td>CHAR(11)</td>
<td>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</td>
</tr>
<tr>
<td>RDB$DELETE_RULE</td>
<td>CHAR(11)</td>
<td>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</td>
</tr>
</tbody>
</table>

### 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>
### Column Name | Data Type | Description
--- | --- | ---
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 table or view.
RDB$EXTERNAL_FILE | VARCHAR(255) | The full path to the external data file if the table is defined with the EXTERNAL FILE clause.
RDB$RUNTIME | BLOB | Table metadata description, used internally for optimization.
RDB$EXTERNAL_DESCRIPTION | BLOB | Could store comments related to the external file of an external table.
RDB$OWNER_NAME | CHAR(31) | The user name of the user who created the table or view originally.
RDB$DEFAULT_CLASS | CHAR(31) | Default security class, used when a new column is added to the table.
RDB$FLAGS | SMALLINT | Internal flags.
RDB$RELATION_TYPE | SMALLINT | 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)

### RDB$RELATION_CONSTRAINTS

RDB$RELATION_CONSTRAINTS stores the definitions of all table-level constraints: primary, unique, foreign key, CHECK, NOT NULL constraints.

| Column Name | Data Type | Description |
--- | --- | ---
RDB$CONSTRAINT_NAME | CHAR(31) | The name of the table-level constraint defined by the user, or otherwise automatically generated by the system.
RDB$CONSTRAINT_TYPE | CHAR(11) | The name of the constraint type: PRIMARY KEY, UNIQUE, FOREIGN KEY, CHECK or NOT NULL.
RDB$RELATION_NAME | CHAR(31) | The name of the table this constraint applies to.
RDB$DEFERRABLE | CHAR(3) | Currently NO in all cases: Firebird does not yet support deferrable constraints.
### RDB$INITIALLY_DEFERRED
- **Column Name**: RDB$INITIALLY_DEFERRED
- **Data Type**: CHAR(3)
- **Description**: Currently NO in all cases

### RDB$INDEX_NAME
- **Column Name**: RDB$INDEX_NAME
- **Data Type**: CHAR(31)
- **Description**: The name of the index that supports this constraint. For a CHECK or a NOT NULL constraint, it is NULL.

## 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>
</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_CLASS

RDB$SECURITY_CLASS 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>
</tbody>
</table>
RDB$DESCRIPTION

Could store comments related to the security class.

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 isc_prepare_transaction2, 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 (“database trigger”)</td>
</tr>
<tr>
<td>RDB$TRIGGER_SEQUENCE</td>
<td>SMALLINT</td>
<td>Position of this trigger in the sequence. Zero usually means that no sequence position is specified</td>
</tr>
<tr>
<td>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RDB$TRIGGER_TYPE</td>
<td>SMALLINT</td>
<td>The event the trigger fires on:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - before insert</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - after insert</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 - before update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 - after update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 - before delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 - after delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17 - before insert or update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18 - after insert or update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25 - before insert or delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>26 - after insert or delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>27 - before update or delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>28 - after update or delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>113 - before insert or update or delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>114 - after insert or update or delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8192 - on connect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8193 - on disconnect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8194 - on transaction start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8195 - on transaction commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8196 - on transaction rollback</td>
</tr>
<tr>
<td>Identification of the exact RDB$TRIGGER_TYPE code is a little more complicated, since it is a bitmap, calculated according to which phase and events are covered and the order in which they are defined. For the curious, the calculation is explained in <a href="https://example.com">this code comment by Mark Rotteveel</a>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDB$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$DEBUG_INFO</td>
<td>BLOB</td>
<td>Contains debugging information about variables used in the trigger</td>
</tr>
</tbody>
</table>
## RDB$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$FIELD_NAME</td>
<td>CHAR(31)</td>
<td>Enumerated type name. Each type name masters its own set of types, e.g., object types, data types, character sets, trigger types, blob subtypes, etc.</td>
</tr>
<tr>
<td>RDB$TYPE</td>
<td>SMALLINT</td>
<td>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 ...</td>
</tr>
<tr>
<td>RDB$TYPE_NAME</td>
<td>CHAR(31)</td>
<td>The name of a member of an enumerated type, e.g., TABLE, VIEW, TRIGGER, etc. in the example above. In the RDB$CHARACTER_SET enumerated type, RDB$TYPE_NAME stores the names of the character sets.</td>
</tr>
<tr>
<td>RDB$DESCRIPTION</td>
<td>BLOB TEXT</td>
<td>Any text comments related to the enumerated type</td>
</tr>
</tbody>
</table>
### Column Name | Data Type | Description
--- | --- | ---
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>RDB$PRIVILEGE</td>
<td>CHAR(6)</td>
<td>The privilege granted hereby:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A - all (all privileges)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S - select (selecting data)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I - insert (inserting rows)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D - delete (deleting rows)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R - references (foreign key)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U - update (updating data)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X - executing (procedure)</td>
</tr>
<tr>
<td>RDB$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$USER_TYPE</td>
<td>SMALLINT</td>
<td>Identifies the type of user the privilege is granted TO (a user, a procedure, a view, etc.)</td>
</tr>
<tr>
<td>RDB$OBJECT_TYPE</td>
<td>SMALLINT</td>
<td>Identifies the type of the object the privilege is granted ON</td>
</tr>
</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$RELATION_NAME</td>
<td>CHAR(31)</td>
<td>The name of the table the view references</td>
</tr>
<tr>
<td>RDB$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>
</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

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$ATTACHMENT_ID</td>
<td>INTEGER</td>
<td>Connection identifier</td>
</tr>
<tr>
<td>MON$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 was specified when the connection was established, the field contains the text 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$CHARACTER_SET in system table RDB$TYPES)</td>
</tr>
<tr>
<td>MON$TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>The date and time when the connection was started</td>
</tr>
</tbody>
</table>
### MON$GARBAGE_COLLECTION

**Column Name:** MON$GARBAGE_COLLECTION  
**Data Type:** SMALLINT  
**Description:** Garbage collection flag (as specified in the attachment’s DPB): 1=allowed, 0=not allowed

### MON$REMOTE_PROCESS

**Column Name:** MON$REMOTE_PROCESS  
**Data Type:** VARCHAR(255)  
**Description:** The full file name and path to the executable file that established this connection

### MON$STAT_ID

**Column Name:** MON$STAT_ID  
**Data Type:** INTEGER  
**Description:** Statistics identifier

---

### 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
- Later attempts to use this connection (i.e., use its handle in API calls) will return errors

**Example**

Closing all connections except for your own (current):

```
DELETE FROM MON$ATTACHMENTS
WHERE MON$ATTACHMENT_ID <> CURRENT_CONNECTION
```

---

### MON$CALL_STACK

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$CALL_ID</td>
<td>INTEGER</td>
<td>Call identifier</td>
</tr>
<tr>
<td>MON$STATEMENT_ID</td>
<td>INTEGER</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>INTEGER</td>
<td>The identifier of the calling trigger or stored procedure</td>
</tr>
</tbody>
</table>
**Appendix E: Monitoring Tables**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>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>
</tbody>
</table>

**EXECUTE STATEMENT Calls**

Information about calls during the execution of the EXECUTE STATEMENT statement does not get into the call stack.

**Example using MON$CALL_STACK**

Getting the call stack for all connections except own:

```sql
WITH RECURSIVE
  HEAD AS (
    SELECT
      CALL.MON$STATEMENT_ID, CALL.MON$CALL_ID,
      CALL.MON$OBJECT_NAME, CALL.MON$OBJECT_TYPE
    FROM MON$CALL_STACK CALL
    WHERE CALL.MON$CALLER_ID IS NULL
    UNION ALL
    SELECT
      CALL.MON$STATEMENT_ID, CALL.MON$CALL_ID,
      CALL.MON$OBJECT_NAME, CALL.MON$OBJECT_TYPE
    FROM MON$CALL_STACK CALL
    JOIN HEAD ON CALL.MON$CALLER_ID = HEAD.MON$CALL_ID
  )

SELECT MON$ATTACHMENT_ID, MON$OBJECT_NAME, MON$OBJECT_TYPE
FROM HEAD
JOIN MON$STATEMENTS STMNT ON STMNT.MON$STATEMENT_ID = HEAD.MON$STATEMENT_ID
WHERE STMNT.MON$ATTACHMENT_ID <> CURRENT_CONNECTION
```
**MON$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$ATTACHMENT_ID</td>
<td>INTEGER</td>
<td>Connection identifier. It contains a valid value only for a connection-level context variable. For transaction-level variables it is NULL.</td>
</tr>
<tr>
<td>MON$TRANSACTION_ID</td>
<td>INTEGER</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(255)</td>
<td>Context variable value</td>
</tr>
</tbody>
</table>

**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$OLDEST_TRANSACTION</td>
<td>INTEGER</td>
<td>The number of the oldest [interesting] transaction (OIT)</td>
</tr>
<tr>
<td>MON$OLDEST_ACTIVE</td>
<td>INTEGER</td>
<td>The number of the oldest active transaction (OAT)</td>
</tr>
<tr>
<td>MON$OLDEST_SNAPSHOT</td>
<td>INTEGER</td>
<td>The number of the transaction that was active at the moment when the OAT was started — oldest snapshot transaction (OST)</td>
</tr>
<tr>
<td>MON$NEXT_TRANSACTION</td>
<td>INTEGER</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>Column Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</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$RESERVE_SPACE</td>
<td>SMALLINT</td>
<td>The flag indicating reserve_space (value 1) or use_all_space (value 0) for filling database pages</td>
</tr>
<tr>
<td>MON$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>
</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$STAT_ID</td>
<td>INTEGER</td>
<td>Statistics identifier</td>
</tr>
</tbody>
</table>
### MON$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 - statement, 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$MAX_MEMORY_USED</td>
<td>BIGINT</td>
<td>The maximum number of bytes used by this object</td>
</tr>
</tbody>
</table>
Not all records in this table have non-zero values. `MON$DATABASE` and objects related to memory allocation have non-zero values. Minor memory allocations are not accrued here but are added to the database memory pool instead.

**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$RECORD_SEQ_READS</td>
<td>BIGINT</td>
<td>Count of records read sequentially</td>
</tr>
<tr>
<td>MON$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>
</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$STATEMENT_ID</td>
<td>INTEGER</td>
<td>Statement identifier</td>
</tr>
<tr>
<td>MON$ATTACHMENT_ID</td>
<td>INTEGER</td>
<td>Connection identifier</td>
</tr>
<tr>
<td>MON$TRANSACTION_ID</td>
<td>INTEGER</td>
<td>Transaction identifier</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.

**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 query.

**Notes**

- 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

**Example**

Cancelling all active queries for the specified connection:

```
DELETE FROM MON$STATEMENTS
WHERE MON$ATTACHMENT_ID = 32
```

**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$TRANSACTION_ID</td>
<td>INTEGER</td>
<td>Transaction identifier (number)</td>
</tr>
<tr>
<td>MON$ATTACHMENT_ID</td>
<td>INTEGER</td>
<td>Connection 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>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$TOP_TRANSACTION</td>
<td>INTEGER</td>
<td>Top-level transaction identifier (number)</td>
</tr>
<tr>
<td>MON$OLDEST_TRANSACTION</td>
<td>INTEGER</td>
<td>Transaction ID of the oldest [interesting] transaction (OIT)</td>
</tr>
<tr>
<td>MON$OLDEST_ACTIVE</td>
<td>INTEGER</td>
<td>Transaction ID of the oldest active transaction (OAT)</td>
</tr>
<tr>
<td>MON$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>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(value 0)</td>
</tr>
<tr>
<td>MON$AUTO_COMMIT</td>
<td>SMALLINT</td>
<td>Flag indicating whether automatic commit is used for the transaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(value 1) or not (value 0)</td>
</tr>
<tr>
<td>MON$AUTO_UNDO</td>
<td>SMALLINT</td>
<td>Flag indicating whether the logging mechanism automatic undo is used for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>
### Appendix F: Character Sets and Collation Sequences

#### Table 189. 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_SWEDFIN</td>
<td>Swedish / Finnish Paradox</td>
</tr>
<tr>
<td></td>
<td>//</td>
<td></td>
<td>PDOX_INTL</td>
<td>International English 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>〃</td>
<td>〃</td>
<td>〃</td>
<td>DB_NLD850</td>
<td>Dutch</td>
</tr>
<tr>
<td>〃</td>
<td>〃</td>
<td>〃</td>
<td>DB_PT850</td>
<td>Portuguese-Brazil</td>
</tr>
<tr>
<td>〃</td>
<td>〃</td>
<td>〃</td>
<td>DB_SVE850</td>
<td>Swedish</td>
</tr>
<tr>
<td>〃</td>
<td>〃</td>
<td>〃</td>
<td>DB_UK850</td>
<td>English-Great Britain</td>
</tr>
<tr>
<td>〃</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>〃</td>
<td>〃</td>
<td>〃</td>
<td>DB_CSY</td>
<td>Czech dBase</td>
</tr>
<tr>
<td>〃</td>
<td>〃</td>
<td>〃</td>
<td>DB_PLK</td>
<td>Polish dBase</td>
</tr>
<tr>
<td>〃</td>
<td>〃</td>
<td>〃</td>
<td>DB_SLO</td>
<td>Slovene dBase</td>
</tr>
<tr>
<td>〃</td>
<td>〃</td>
<td>〃</td>
<td>PDOX_CSY</td>
<td>Czech Paradox</td>
</tr>
<tr>
<td>〃</td>
<td>〃</td>
<td>〃</td>
<td>PDOX_HUN</td>
<td>Hungarian Paradox</td>
</tr>
<tr>
<td>〃</td>
<td>〃</td>
<td>〃</td>
<td>PDOX_PLK</td>
<td>Polish Paradox</td>
</tr>
<tr>
<td>〃</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>〃</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>〃</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>〃</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>〃</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>〃</td>
<td>〃</td>
<td>〃</td>
<td>DB_DAN865</td>
<td>Danish dBase</td>
</tr>
<tr>
<td>〃</td>
<td>〃</td>
<td>〃</td>
<td>DB_NOR865</td>
<td>Norwegian dBase</td>
</tr>
<tr>
<td>〃</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>
</tbody>
</table>

Appendix F: 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>GB_2312</td>
<td>57</td>
<td>2</td>
<td>GB_2312</td>
<td>Simplified Chinese (Hong Kong, Korea)</td>
</tr>
<tr>
<td>GB18030</td>
<td>69</td>
<td>4</td>
<td>GB18030</td>
<td>Chinese</td>
</tr>
<tr>
<td>GBK</td>
<td>67</td>
<td>2</td>
<td>GBK</td>
<td>Chinese</td>
</tr>
<tr>
<td>GBK_UNICODE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISO8859_1</td>
<td>21</td>
<td>1</td>
<td>ISO8859_1</td>
<td>Latin I</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>DE_DE</td>
<td></td>
<td></td>
<td></td>
<td>German</td>
</tr>
<tr>
<td>DU_NL</td>
<td></td>
<td></td>
<td></td>
<td>Dutch</td>
</tr>
<tr>
<td>EN_UK</td>
<td></td>
<td></td>
<td></td>
<td>English-Great Britain</td>
</tr>
<tr>
<td>EN_US</td>
<td></td>
<td></td>
<td></td>
<td>U.S. English</td>
</tr>
<tr>
<td>ES_ES</td>
<td></td>
<td></td>
<td></td>
<td>Spanish</td>
</tr>
<tr>
<td>ES_ES_CI_AI</td>
<td></td>
<td></td>
<td></td>
<td>Spanish — case insensitive and + accent-insensitive</td>
</tr>
<tr>
<td>FI_FI</td>
<td></td>
<td></td>
<td></td>
<td>Finnish</td>
</tr>
<tr>
<td>FR_CA</td>
<td></td>
<td></td>
<td></td>
<td>French-Canada</td>
</tr>
<tr>
<td>FR_FR</td>
<td></td>
<td></td>
<td></td>
<td>French</td>
</tr>
<tr>
<td>FR_FR_CI_AI</td>
<td></td>
<td></td>
<td></td>
<td>French — case insensitive + accent insensitive</td>
</tr>
<tr>
<td>IS_IS</td>
<td></td>
<td></td>
<td></td>
<td>Icelandic</td>
</tr>
<tr>
<td>IT_IT</td>
<td></td>
<td></td>
<td></td>
<td>Italian</td>
</tr>
<tr>
<td>NO_NO</td>
<td></td>
<td></td>
<td></td>
<td>Norwegian</td>
</tr>
<tr>
<td>PT_PT</td>
<td></td>
<td></td>
<td></td>
<td>Portuguese</td>
</tr>
<tr>
<td>PT_BR</td>
<td></td>
<td></td>
<td></td>
<td>Portuguese-Brazil</td>
</tr>
<tr>
<td>SV_SV</td>
<td></td>
<td></td>
<td></td>
<td>Swedish</td>
</tr>
<tr>
<td>CS_CZ</td>
<td></td>
<td></td>
<td></td>
<td>Czech</td>
</tr>
<tr>
<td>ISO_HUN</td>
<td></td>
<td></td>
<td></td>
<td>Hungarian — case insensitive, accent sensitive</td>
</tr>
<tr>
<td>ISO_PLK</td>
<td></td>
<td></td>
<td></td>
<td>Polish</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>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>NEXT</td>
<td></td>
<td></td>
<td>NEXT_DEU</td>
<td>German</td>
</tr>
<tr>
<td>NEXT</td>
<td></td>
<td></td>
<td>NXT_ESP</td>
<td>Spanish</td>
</tr>
<tr>
<td>NEXT</td>
<td></td>
<td></td>
<td>NXT_FRA</td>
<td>French</td>
</tr>
<tr>
<td>NEXT</td>
<td></td>
<td></td>
<td>NXT_ITA</td>
<td>Italian</td>
</tr>
<tr>
<td>NEXT</td>
<td>19</td>
<td>1</td>
<td>NXT_US</td>
<td>U.S. English</td>
</tr>
<tr>
<td>NONE</td>
<td>0</td>
<td>1</td>
<td>NONE</td>
<td>Neutral code page. Translation to upper case is performed only for code ASCII 97-122. Recommendation: avoid this character set</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>UNICODE_FSS</td>
<td>3</td>
<td>3</td>
<td>UNICODE_FSS</td>
<td>All English</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>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>USC_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_HUND C</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</td>
<td>Czech — Case insensitive</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>WIN1251_UA</td>
<td>Ukrainian</td>
</tr>
<tr>
<td>//</td>
<td>//</td>
<td>//</td>
<td>PXW_CYRL</td>
<td>Paradox Cyrillic (Russian)</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>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>WIN1253</td>
<td>54</td>
<td>1</td>
<td>WIN1253</td>
<td>ANSI Greek</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PXW_GREEK</td>
<td>Paradox Greek</td>
</tr>
<tr>
<td>WIN1254</td>
<td>55</td>
<td>1</td>
<td>WIN1254</td>
<td>ANSI Turkish</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PXW_TURK</td>
<td>Paradox 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></td>
<td></td>
<td></td>
<td>WIN1257_EE</td>
<td>Estonian — Dictionary ordering</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WIN1257_LT</td>
<td>Lithuanian — Dictionary ordering</td>
</tr>
<tr>
<td></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 G: 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 2.5 Language Reference.

The Initial Writers of the Original Documentation are: Paul Vinkenoog, Dmitry Yemanov and Thomas Woinke. Writers of text originally in Russian are Denis Simonov, Dmitry Filippov, Alexander Karpeykin, Alexey Kovyazin and Dmitry Kuzmenko.

Copyright © 2008-2020. 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 © 2001-2020 by their respective authors. All Rights Reserved.

Contributor(s): Mark Rotteveel.

Portions created by Mark Rotteveel are Copyright © 2018-2020. All Rights Reserved. (Contributor contact(s): mrotteveel at users dot sourceforge dot net).
# Appendix H: 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>Version</th>
<th>Date</th>
<th>Type</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.9</td>
<td>16 Oct 2021</td>
<td>MR</td>
<td>EXECUTE STATEMENT named parameters are regular identifiers (<a href="https://github.com/FirebirdSQL/firebird-documentation/issues/164">#164</a>)</td>
</tr>
<tr>
<td>1.8</td>
<td>31 Jul 2021</td>
<td>MR</td>
<td>Fix behaviour documented for SNAPSHOT TABLE STABILITY (<a href="https://github.com/FirebirdSQL/firebird-documentation/issues/158">#158</a>)</td>
</tr>
<tr>
<td>1.7</td>
<td>23 Jul 2021</td>
<td>MR</td>
<td>Remove extra SELECT in select syntax</td>
</tr>
<tr>
<td>1.6</td>
<td>13 Jun 2021</td>
<td>MR</td>
<td>• Fixed wrong table title NUMERIC → DECIMAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Fixed wrong link title DATEADD → DATEDIFF</td>
</tr>
<tr>
<td>1.5</td>
<td>27 Apr 2021</td>
<td>MR</td>
<td>• Added missing } in regular expression special characters (see <a href="https://github.com/FirebirdSQL/firebird-documentation/issues/124">issue 124</a>)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Fixed rendering issue hiding the _ in regular expression special characters</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Improve wording of CURRENT_CONNECTION and CURRENT_TRANSACTION (see <a href="https://github.com/FirebirdSQL/firebird-documentation/issues/96">issue 96</a>)</td>
</tr>
<tr>
<td>1.4</td>
<td>27 Apr 2021</td>
<td>MR</td>
<td>Fixed GRANT syntax (see <a href="https://github.com/FirebirdSQL/firebird-documentation/issues/130">issue 130</a>)</td>
</tr>
<tr>
<td>1.3</td>
<td>13 Jun 2020</td>
<td>MR</td>
<td>Various minor changes in copy and styling while checking migrating the German version to AsciiDoc</td>
</tr>
<tr>
<td>1.2</td>
<td>6 Jun 2020</td>
<td>MR</td>
<td>• Split chapter Built-in Functions and Context Variables into two chapters: Built-in Functions, and Context Variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Removed incorrect caution on expressions in ORDER BY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Made function and context variable description use consistent order</td>
</tr>
<tr>
<td>1.1</td>
<td>1 Jun 2020</td>
<td>MR</td>
<td>Conversion to AsciiDoc. Various copy-editing, and fixing factual errors and other problems while checking and correcting AsciiDoc.</td>
</tr>
</tbody>
</table>
## Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Author</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>22 Jan 2018</td>
<td>H.E.M. B.</td>
<td>Updated the file history link at the top of this chapter to reflect the migration of the Doc source tree to Github. Typos fixed/updates by M. Rotteveel Dec. 2017/Jan. 2018</td>
</tr>
<tr>
<td>1</td>
<td>11 Aug 2017</td>
<td>H.E.M. B.</td>
<td>There have been no further changes to the content during the final review period. The chapters DML, PSQL, Functions and Variables, Transactions and Security were not reviewed in this phase.</td>
</tr>
<tr>
<td>0.90</td>
<td>11 Aug 2016</td>
<td>H.E.M. B.</td>
<td>Several revisions were published during the year, as a Beta 1, with reviews of various sections by Paul Vinkenoog, Aage Johansen and Mark Rotteveel. This revision (0.906) awaits final revision of some later sections, marked as “Editor’s Note” in red italics.</td>
</tr>
<tr>
<td>0.90</td>
<td>1 Sep 2015</td>
<td>H.E.M. B.</td>
<td>Original was in Russian, translated by Dmitry Borodin (MegaTranslations). Raw translation edited and converted to DocBook, as this revision (0.900), by Helen Borrie. This revision distributed as a PDF build only, for review by Dmitry Yemanov, et al. Reviewers, please pay attention to the comments like this: Editor’s note :: The sky is falling, take cover!</td>
</tr>
</tbody>
</table>