Firebird SQL best practices

Review of some SQL features available and that people often forget about

Author: Philippe Makowski IBPhoenix
Email: pmakowski@ibphoenix.com

Date: 2016-09-29

Common table expression

Syntax

```
WITH [RECURSIVE]  -- new keywords
CTE_A -- first table expression’s name
 [(a1, a2, ...)] -- fields aliases, optional
   AS ( SELECT ... ), -- table expression’s definition
CTE_B -- second table expression
 [(b1, b2, ...)]
   AS ( SELECT ... ),
...
SELECT ...  -- main query, used both
FROM CTE_A, CTE_B, -- table expressions
TAB1, TAB2  -- and regular tables
WHERE ...
```

Emulate loose index scan

The term "loose indexscan" is used in some other databases for the operation of using a btree index to retrieve the distinct values of a column efficiently; rather than scanning all equal values of a key, as soon as a new value is found, restart the search by looking for a larger value. This is much faster when the index has many equal keys.

A table with 10,000,000 rows, and only 3 differents values in row.

```
CREATE TABLE HASH
(
  ID                          INTEGER          NOT NULL,
  SMALLDISTINCT               SMALLINT,
 PRIMARY KEY (ID)
);
CREATE ASC INDEX SMALLDISTINCT_IDX ON HASH (SMALLDISTINCT);
```

Without CTE :

```
SELECT DISTINCT SMALLDISTINCT FROM HASH

SMALLDISTINCT
=============
0
1
2

PLAN SORT ((HASH NATURAL))
Prepared in 0.001 sec, processed in 13.234 sec
HASH 10000000 Non-Indexed reads
```

Emulate loose index scan with recursive CTE :

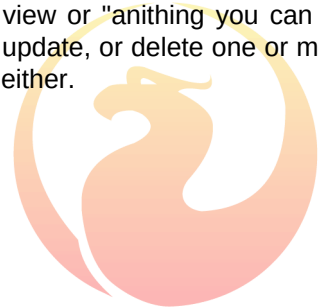
```
WITH RECURSIVE
t AS (SELECT min(smalldistinct) AS smalldistinct FROM HASH
      UNION ALL
      SELECT (SELECT min(smalldistinct) FROM HASH
              WHERE smalldistinct > t.smalldistinct)
       FROM t WHERE t.smalldistinct IS NOT NULL)
SELECT smalldistinct FROM t WHERE smalldistinct IS NOT NULL
UNION ALL
SELECT NULL FROM RDB$DATABASE
WHERE EXISTS(SELECT 1 FROM HASH WHERE smalldistinct IS NULL)

PLAN (T HASH ORDER SMALLDISTINCT_IDX INDEX (SMALLDISTINCT_IDX))
PLAN (HASH INDEX (SMALLDISTINCT_IDX))
Prepared in 0.001 sec, processed in 3.312 sec
HASH 3 Indexed reads
RDB$DATABASE 1 Non-Indexed read
```

MERGE

The purpose of MERGE is to read data from the source and INSERT, UPDATE or DELETE in the target table according to a condition.

The source may be table, a view or "anithing you can select from" in general. Each source record will be used to update, or delete one or more target record, insert a new record in the target table, or neither.



Example for MERGE

```
create table stock ( item_id int not null primary key, balance int);
insert into stock values (10, 2200);
insert into stock values (20, 1900);
commit;
select * from stock;

   ITEM_ID      BALANCE
========== ============
        10         2200
        20         1900
```

```
create table buy ( item_id int not null primary key, volume int);
insert into buy values (10, 1000);
insert into buy values (30, 300);
commit;
select * from buy;

   ITEM_ID       VOLUME
========== ============
        10         1000
        30          300
```

```
create table sale ( item_id int not null primary key, volume int);
insert into sale values (10, 2200);
insert into sale values (20, 1000);
commit;
select * from sale;

   ITEM_ID       VOLUME
========== ============
        10         2200
        20         1000
```

Update the stock with what we bought.

```
select * from stock;
   ITEM_ID      BALANCE
========== ============
        10         2200
        20         1900

MERGE INTO stock USING buy ON stock.item_id = buy.item_id
 WHEN MATCHED THEN UPDATE SET balance = balance + buy.volume
 WHEN NOT MATCHED THEN INSERT VALUES (buy.item_id, buy.volume);
SELECT * FROM stock ORDER BY item_id;
   ITEM_ID      BALANCE
========== ============
        10         3200
        20         1900
        30          300
```

Then update the stock with what we sale.

```
SELECT * FROM stock ORDER BY item_id;
   ITEM_ID      BALANCE
========== ============
        10         3200
        20         1900
        30          300

MERGE INTO stock USING sale ON stock.item_id = sale.item_id
 WHEN MATCHED AND balance - volume > 0 THEN UPDATE SET balance = balance - volume
 WHEN MATCHED THEN DELETE;
SELECT * FROM stock ORDER BY item_id;
   ITEM_ID      BALANCE
========== ============
        10         1000
        20          900
        30          300
```

See the DELETE in action :

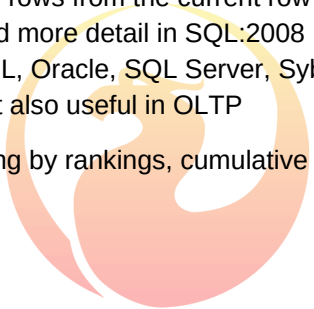
```
rollback;
SELECT * FROM stock ORDER BY item_id;
   ITEM_ID      BALANCE
========== ============
        10         2200
        20         1900

select * from sale;
   ITEM_ID       VOLUME
========== ============
        10         2200
        20         1000

MERGE INTO stock USING sale ON stock.item_id = sale.item_id
 WHEN MATCHED AND balance - volume > 0 THEN UPDATE SET balance = balance - volume
 WHEN MATCHED THEN DELETE;
SELECT * FROM stock ORDER BY item_id;
   ITEM_ID      BALANCE
========== ============
        20          900
```

What are Windowing Functions?

- Similar to classical aggregates but does more!
- Provides access to set of rows from the current row
- Introduced SQL:2003 and more detail in SQL:2008
- Supported by PostgreSQL, Oracle, SQL Server, Sybase and DB2
- Used in OLAP mainly but also useful in OLTP
    - Analysis and reporting by rankings, cumulative aggregates



Windowed Table Functions

- Windowed table function
  - operates on a window of a table
  - returns a value for every row in that window
  - the value is calculated by taking into consideration values from the set of rows in that window
- 8 new windowed table functions
- In addition, old aggregate functions can also be used as windowed table functions
- Allows calculation of moving and cumulative aggregate values.

A Window

- Represents set of rows that is used to compute additionnal attributes
- Based on three main concepts
  - **partition**
    - specified by PARTITION BY clause in OVER()
    - Allows to subdivide the table, much like GROUP BY clause
    - Without a PARTITION BY clause, the whole table is in a single partition
  - **order**
    - defines an order with a partition
    - may contain multiple order items
      - Each item includes a value-expression
      - NULLS FIRST/LAST defines ordering semantics for NULL
    - this clause is independant of the query's ORDER BY clause

- **frame** (Firebird don't implement frame yet)

Built-in Windowing Functions

- RANK () OVER ...
- DENSE_RANK () OVER ...
- LAG () OVER ...
- LEAD () OVER ...
- ROW_NUMBER () OVER ...
- FIRST_VALUE () OVER ...
- LAST_VALUES () OVER ...
- NTH_VALUE () OVER ...

Set Functions as Window Functions

Who are the highest paid relatively compared with the department average?

```
select emp_no, dept_no, salary,
  avg(salary) over (partition by dept_no) as dept_avg,
  salary - avg(salary) over (partition by dept_no) as diff
from employee
order by diff desc;
```

EMP_NO	DEPT_NO	SALARY	DEPT_AVG	DIFF
118	115	7480000.00	6740000.00	740000.00
105	000	212850.00	133321.50	79528.50
107	670	111262.50	71268.75	39993.75
2	600	105900.00	66450.00	39450.00
85	100	111262.50	77631.25	33631.25
4	621	97500.00	69184.87	28315.13
46	900	116100.00	92791.31	23308.69
9	622	75060.00	53409.16	21650.84

Performance

List orders, quantity ordered and cumulative quantity ordered by day

ORDER_DATE	PO_NUMBER	QTY_ORDERED	QTY_CUMUL_DAY
1991-03-04	V91E0210	10	10
1992-07-26	V92J1003	15	15
1992-10-15	V92E0340	7	7
1992-10-15	V92F3004	3	10
1993-02-03	V9333005	2	2
1993-03-22	V93C0120	1	1
1993-04-27	V9333006	5	5
1993-08-01	V93H3009	3	3
1993-08-09	V9324200	1000	1000
1993-08-09	V93C0990	40	1040

Without window function

```
SELECT ORDER_DATE, CUST_NO,QTY_ORDERED,
      (SELECT SUM(QTY_ORDERED)
       FROM   SALES AS Si
       WHERE  Si.ORDER_DATE = S.ORDER_DATE
       AND Si.CUST_NO <= S.CUST_NO)
      AS QTY_CUMUL_DAY
FROM   SALES AS S
ORDER BY S.ORDER_DATE, S.CUST_NO
```

```
PLAN (SI INDEX (RDB$FOREIGN25))
PLAN SORT (S NATURAL)
SALES 591 indexed reads
SALES  33 non indexed reads
```

With window function

```
SELECT ORDER_DATE, PO_NUMBER,QTY_ORDERED,
      SUM(QTY_ORDERED)
      OVER (PARTITION BY ORDER_DATE
       ORDER BY PO_NUMBER)
      AS QTY_CUMUL_DAY
FROM   SALES
ORDER BY ORDER_DATE, PO_NUMBER
```

```
PLAN SORT (SALES NATURAL)
SALES 33 non indexed reads
```

And you can extend it nearly without cost

```
SELECT ORDER_DATE, PO_NUMBER,QTY_ORDERED,
      SUM(QTY_ORDERED)
      OVER (PARTITION BY ORDER_DATE
       ORDER BY PO_NUMBER)
      AS QTY_CUMUL_DAY,
      SUM(QTY_ORDERED)
      OVER (PARTITION BY  EXTRACT(YEAR FROM ORDER_DATE),EXTRACT(MONTH FROM ORDER_DATE)
       ORDER BY  ORDER_DATE, PO_NUMBER)
      AS QTY_CUMUL_MONTH,
      SUM(QTY_ORDERED)
      OVER (PARTITION BY  EXTRACT(YEAR FROM ORDER_DATE)
       ORDER BY  ORDER_DATE, PO_NUMBER)
      AS QTY_CUMUL_YEAR
FROM   SALES
ORDER BY ORDER_DATE, PO_NUMBER
```

```
PLAN SORT (SALES NATURAL)
SALES 33 non indexed reads
```

ORDER_DATE	PO_NUMBER	QTY_ORDERED	QTY_CUMUL_DAY	QTY_CUMUL_MONTH	QTY_CUMUL_YEAR
1991-03-04	V91E0210	10	10	10	10
1992-07-26	V92J1003	15	15	15	15
1992-10-15	V92E0340	7	7	7	22
1992-10-15	V92F3004	3	10	10	25
1993-02-03	V9333005	2	2	2	2
1993-03-22	V93C0120	1	1	1	3
1993-04-27	V9333006	5	5	5	8
1993-08-01	V93H3009	3	3	3	11
1993-08-09	V9324200	1000	1000	1003	1011
1993-08-09	V93C0990	40	1040	1043	1051
1993-08-16	V9324320	1	1	1044	1052
1993-08-20	V93J3100	16	16	1060	1068
1993-08-27	V93F3088	10	10	1070	1078

Thank you !