Firebird SQL best practices

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

Author: Philippe Makowski IBPhoenix
Email: pmakowski@ibphoenix.com
Licence: Public Documentation License
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 index scan" 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 different values in row.

```sql
CREATE TABLE HASH
(
    ID                      INTEGER         NOT NULL,
    SMALLDISTINCT           SMALLINT,
    PRIMARY KEY (ID)
);
CREATE ASC INDEX SMALLDISTINCT_IDX ON HASH (SMALLDISTINCT);
```
Without CTE:

```sql
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
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 "anything 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

```sql
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;
```

<table>
<thead>
<tr>
<th>ITEM_ID</th>
<th>BALANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2200</td>
</tr>
<tr>
<td>20</td>
<td>1900</td>
</tr>
</tbody>
</table>
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;

<table>
<thead>
<tr>
<th>ITEM_ID</th>
<th>VOLUME</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1000</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
</tr>
</tbody>
</table>
```sql
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;
```

<table>
<thead>
<tr>
<th>ITEM_ID</th>
<th>VOLUME</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2200</td>
</tr>
<tr>
<td>20</td>
<td>1000</td>
</tr>
</tbody>
</table>
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;

<table>
<thead>
<tr>
<th>ITEM_ID</th>
<th>BALANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3200</td>
</tr>
<tr>
<td>20</td>
<td>1900</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>ITEM_ID</th>
<th>BALANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1000</td>
</tr>
<tr>
<td>20</td>
<td>900</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
</tr>
</tbody>
</table>
```
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 additional 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 independent 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 ...
Who are the highest paid relatively compared with the department average?

```sql
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;
```

<table>
<thead>
<tr>
<th>EMP_NO</th>
<th>DEPT_NO</th>
<th>SALARY</th>
<th>DEPT_AVG</th>
<th>DIFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>118</td>
<td>115</td>
<td>7480000.00</td>
<td>6740000.00</td>
<td>740000.00</td>
</tr>
<tr>
<td>105</td>
<td>000</td>
<td>212850.00</td>
<td>133321.50</td>
<td>79528.50</td>
</tr>
<tr>
<td>107</td>
<td>670</td>
<td>111262.50</td>
<td>71268.75</td>
<td>39993.75</td>
</tr>
<tr>
<td>2</td>
<td>600</td>
<td>105900.00</td>
<td>66450.00</td>
<td>39450.00</td>
</tr>
<tr>
<td>85</td>
<td>100</td>
<td>111262.50</td>
<td>77631.25</td>
<td>33631.25</td>
</tr>
<tr>
<td>4</td>
<td>621</td>
<td>97500.00</td>
<td>69184.87</td>
<td>28315.13</td>
</tr>
<tr>
<td>46</td>
<td>900</td>
<td>116100.00</td>
<td>92791.31</td>
<td>23308.69</td>
</tr>
<tr>
<td>9</td>
<td>622</td>
<td>75060.00</td>
<td>53409.16</td>
<td>21650.84</td>
</tr>
</tbody>
</table>
List orders, quantity ordered and cumulative quantity ordered by day

<table>
<thead>
<tr>
<th>ORDER_DATE</th>
<th>PO_NUMBER</th>
<th>QTY_ORDERED</th>
<th>QTY_CUMUL_DAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1991-03-04</td>
<td>V91E0210</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1992-07-26</td>
<td>V92J1003</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>1992-10-15</td>
<td>V92E0340</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1992-10-15</td>
<td>V92F3004</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>1993-02-03</td>
<td>V9333005</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1993-03-22</td>
<td>V93C0120</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1993-04-27</td>
<td>V9333006</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1993-08-01</td>
<td>V93H3009</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1993-08-09</td>
<td>V9324200</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>1993-08-09</td>
<td>V93C0990</td>
<td>40</td>
<td>1040</td>
</tr>
</tbody>
</table>
Without window function

```sql
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
```

```sql
PLAN (SI INDEX (RDB$FOREIGN25))
PLAN SORT (S NATURAL)
SALES 591 indexed reads
SALES 33 non indexed reads
```
**With window function**

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