Data retrieval

The most frequently used operation in transactional databases is the data retrieval operation.

SELECT is used to retrieve zero or more rows from one or more tables in a database. In most applications, SELECT is the most commonly used DML command. In specifying a SELECT query, the user specifies a description of the desired result set, but they do not specify what physical operations must be executed to produce that result set. Translating the query into an optimal query plan is left to the database system, more specifically to the query optimizer.

For those new to Firebird SQL, please refer to the IBExpert article, SQLBasics.

SELECT

The SELECT statement has the following syntax:

Syntax InterBase® 7.1

```
SELECT [TRANSACTION transaction]
     [DISTINCT | ALL]
     {* | val [, val ...]}
     [INTO :var [, :var ...]]
     FROM tableref [, tableref ...]
     [WHERE search_condition]
     [GROUP BY col [COLLATE collation] [, col [COLLATE collation] ...]
     [HAVING search_condition]
     [UNION [ALL] select_expr]
     [PLAN plan_expr]
     [ORDER BY order_list]
     [ROWS value [TO upper_value] [BY step_value][PERCENT][WITH TIES]]
     [FOR UPDATE [OF col [, col ...]]];
```

Description

SELECT retrieves data from tables, views, or stored procedures. Variations of the SELECT statement make it possible to:

- Retrieve a single row, or part of a row, from a table. This operation is referred to as a singleton select. In embedded applications, all SELECT statements that occur outside the context of a cursor must be singleton selects.
- Retrieve multiple rows, or parts of rows, from a table. In embedded applications, multiple row retrieval is accomplished by embedding a SELECT within a DECLARE CURSOR statement. In isql, SELECT can be used directly to retrieve multiple rows.
- Retrieve related rows, or parts of rows, from a join of two or more tables.
- Retrieve all rows, or parts of rows, from union of two or more tables.
- Return portions or sequential portions of a larger result set; useful for Web developers, among

others.
- All SELECT statements consist of two required clauses (SELECT, FROM), and possibly others INTO, WHERE, GROUP BY, HAVING, UNION, PLAN, ORDER BY, ROWS).

Notes on SELECT syntax

- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is 6 characters long:

my_array = varchar(6)[5,5]

Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of integers that begins at 10 and ends at 20: my_array = integer[20:30]

- In SQL and isql, you cannot use val as a parameter placeholder (like "?").
- In DSQL and isql, val cannot be a variable.
- You cannot specify a COLLATE clause for Blob columns.

Important: In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in isql, the semicolon is a terminating symbol for the statement, so it must be included.

Source: InterBase® 7.1 Language Reference Guide

The Firebird syntax deviates slightly from InterBase®:

Syntax Firebird up to 1.5

```
SELECT
   [FIRST (m)] [SKIP (n)] [[ALL] | DISTINCT]
   <list of columns> [, [column-name] | expression | constant ] AS alias-name]
FROM <table-or-procedure-or-view>
   [{[[INNER] | [{LEFT | RIGHT | FULL} [OUTER]] JOIN)] <table-or-procedure-or-view>
   ON <join-conditions [{JOIN ...]]
   [WHERE <search-conditions>]
   [GROUP BY <grouped-column-list>]]
   [HAVING <search-condition>]
   [UNION <select-expression>[ALL]]
   [PLAN <plan_expression>]
   [ORDER BY <column-list>]
   [FOR UPDATE [OF col1 [, col2 ...]][WITH LOCK]];
```

Source: The Firebird Book by Helen Borrie

Syntax Firebird 2.0

```
<select statement> ::=
   <select expression> [FOR UPDATE] [WITH LOCK]

<select expression> ::=
   <query specification> [UNION [{ALL | DISTINCT}] <query specification>]
```

```
<query specification> ::=
   SELECT [FIRST <value>] [SKIP <value>] <select list>
   FROM <table expression list>
   WHERE <search condition>
   GROUP BY <group value list>
   HAVING <group condition>
   PLAN <plan item list>
   ORDER BY <sort value list>
   ROWS <value> [TO <value>]

<table expression> ::=
   <table name> | <joined table> | <derived table>

<joined table> ::=
   {<cross join> | <qualified join>}

<cross join> ::=
   <table expression> CROSS JOIN <table expression>

<qualified join> ::=
   <table expression> [{INNER | {LEFT | RIGHT | FULL} [OUTER]}] JOIN <table
expression>
   ON <join condition>

<derived table> ::=
   '(' <select expression> ')'
```

Conclusions

- FOR UPDATE mode and row locking can only be performed for a final dataset, they cannot be applied to a subquery.
- Unions are allowed inside any subquery.
- Clauses FIRST, SKIP, PLAN, ORDER BY, ROWS are allowed for any subquery.

Notes:

- Either FIRST/SKIP or ROWS is allowed, but a syntax error is thrown if you try to mix the syntaxes.
- An INSERT statement accepts a select expression to define a set to be inserted into a table. Its SELECT part supports all the features defined for select statments/expressions.
- UPDATE and DELETE statements are always based on an implicit cursor iterating through its target table and limited with the WHERE clause. You may also specify the final parts of the select expression syntax to limit the number of affected rows or optimize the statement.

Also new to Firebird 2.0: EXECUTE BLOCK statement - The SQL language extension EXECUTE BLOCK makes “dynamic PSQL” available to SELECT specifications. It has the effect of allowing a self-contained block of PSQL code to be executed in dynamic SQL as if it were a stored procedure. For further information, please refer to EXECUTE BLOCK statement.

Clauses allowed at the end of UPDATE/DELETE statements are PLAN, ORDER BY and ROWS.

Source: Firebird 2.0.4 Release Notes

back to top of page

FIRST (m) SKIP (n)

<FIRST (m) and SKIP (n) are optional keywords, which can be used together or individually. They allow selection and/or the omission of the first m/n rows from the resulting data sets of an ordered set. m and n are integers or simple integer arguments (both without the brackets) or expressions (within brackets) resolving to integers. Logically it should only be used with an ordered set (specified by ORDER BY). If used, these should precede all other specifications.

See also:

- Firebird 2.0 Language Reference Update: FIRST and SKIP
- Firebird Null Guide: Internal functions and directives

DISTINCT

This suppresses all duplicate rows in the output or resulting sets, thus preventing duplicate values from being returned.

ALL

This retrieves every value which meets the specified conditions. It is also the default for the return sets, and so therefore does not need to be explicitly specified.

See also:

Firebird 2.0 Language Reference Update: ALL

FROM

The FROM clause specifies a list of tables, views, and stored procedures (with output arguments) from which to retrieve data. if the query involves joining one that one structure, FROM specifies the leftmost structure. The list then needs to be completed using joins (joins can even be nested). Please refer to JOIN statement for further information.

New to Firebird 2.0: support for derived tables in DSQL (subqueries in FROM clause) as defined by SQL200X. A derived table is a set, derived from a dynamic SELECT statement. Derived tables can be nested, if required, to build complex queries and they can be involved in joins as though they were normal tables or views.

Syntax

```
SELECT
   <select list>
FROM
```

```
   <table reference list>

   <table reference list> ::= <table reference> [{<comma> <table
reference>}...]

   <table reference> ::=
      <table primary>
     | <joined table>

   <table primary> ::=
      <table> [[AS] <correlation name>]
        | <derived table>

   <derived table> ::=
      <query expression> [[AS] <correlation name>]
        [<left paren> <derived column list> <right paren>]

   <derived column list> ::= <column name> [{<comma> <column name>}...]
```

Examples can be found here.

Points to Note

- Every column in the derived table must have a name. Unnamed expressions like constants should be added with an alias or the column list should be used.
- The number of columns in the column list should be the same as the number of columns from the query expression.
- The optimizer can handle a derived table very efficiently. However, if the derived table is involved in an inner join and contains a subquery, then no join order can be made.

back to top of page

WHERE

The WHERE clause is a filter specification, used to define or limit the rows for the return sets or which rows should be forwarded for further processing such as ORDER BY or GROUP BY.

A WHERE clause can also contain its own SELECT statement, referred to as a subquery.

```
<search_conditions> include the following:

<search_condition> = val operator {val | (select_one)}
   | val [NOT] BETWEEN val AND val
   | val [NOT] LIKE val [ESCAPE val]
   | val [NOT] IN (val [, val ...] | select_list)
   | val IS [NOT] NULL
   | val {>= | <=} val
   | val [NOT] {= | < | >} val
   | {ALL | SOME | ANY} (select_list)
   | EXISTS (select_expr)
```

```
    | SINGULAR (select_expr)
    | val [NOT] CONTAINING val
    | val [NOT] STARTING [WITH] val
    | (search_condition)
    | NOT search_condition
    | search_condition OR search_condition
    | search_condition AND search_condition
```

Please refer to Comparison Operators for a full list of valid operators.

Logical conjunctions

AND demands that the conditions to its left and right are both met. e.g.:

```
select *
  from customer
  where (contact_first = 'Glen')
  and (contact_last = 'Brown')
```

OR demands that one of the conditions to its left and right are met, e.g.:

```
select *
  from customer
  where (contact_last = 'Brocket')
  or (contact_last = 'Brown')
```

NOT excludes the condition defined to its right, e.g.:

```
select *
  from customer
  where not (contact_last = 'Brocket')
```

Logical conjunctions can of course also be combined.

GROUP BY

GROUP BY is an optional clause, allowing the resulting sets to be grouped and summarized by common column values into one or more groups, thus aggregating or summarizing the returned data sets. these groupings often include aggregate functions. It is used in conjunction with HAVING.

The group is formed by aggregating (collecting together) all rows where a column named in both the column list and the GROUP BY clause share a common value. The column and/or field specified must of course be groupable, otherwise the query will be rejected. Any NULL values contained in rows in the targeted column are ignored for the aggregation. So if, for example, you wish to calculate averages, you must first consider whether NULL fields should be left out of the calculation, or treated as zero (which entails a little work on the developer side with a #BEFORE or AFTER|BEFORE INSERT trigger).

Firebird 2.0 introduced some useful improvements to SQL sorting operations - please refer to Improvements in sorting in the Firebird 2.0.4. Release Notes for details.

See also:

Firebird 2.0 Language Reference Update: GROUP BY

COLLATE

Specifies the collation order for the data retrieved by the query.

Collation order in a GROUP BY clause: when CHAR or VARCHAR columns are grouped in a SELECT statement, it can be necessary to specify a collation order for the grouping, especially if columns used for grouping use different collation orders.

To specify the collation order to use for grouping columns in the GROUP BY clause, include a COLLATE clause after the column name.

Please note that it is not possible to specify a COLLATE order for Blob columns.

See also:

- Firebird 2.0 Language Reference Update: COLLATE subclause for text BLOB columns
- Collate
- CREATE COLLATION (Firebird 2.1)
- New collations in Firebird 2.1
- New collations in Firebird 2

back to top of page

HAVING

The HAVING condition is optional and may be used together with GROUP BY to specify a condition that limits the grouped rows returned - similar to the WHERE clause. In fact, the HAVING clause can often replace the WHERE clause in a grouping query. Perhaps the simplest way to discern the correct use of these two clauses is to use a WHERE clause to limit rows and a HAVING clause to limit groups. The HAVING clause is applied to the groups after the set has been partitioned. A WHERE filter may still be necessary for the incoming set. To maximize perfomance it is important to use WHERE conditions to pre-filter groups and then use HAVING for filtering on the basis of the results returned (after the grouping has been done) by aggregating functions.

The HAVING clause can use the same arguments as the WHERE clause:

```
<search_conditions> include the following:

<search_condition> = val operator {val | (select_one)}
   | val [NOT] BETWEEN val AND val
   | val [NOT] LIKE val [ESCAPE val]
   | val [NOT] IN (val [, val ...] | select_list)
   | val IS [NOT] NULL
```

```
    | val {>= | <=} val
    | val [NOT] {= | < | >} val
    | {ALL | SOME | ANY} (select_list)
    | EXISTS (select_expr)
    | SINGULAR (select_expr)
    | val [NOT] CONTAINING val
    | val [NOT] STARTING [WITH] val
    | (search_condition)
    | NOT search_condition
    | search_condition OR search_condition
    | search_condition AND search_condition
```

Please refer to Comparison Operators for a full list of valid operators.

See also:

Firebird 2.0 Language Reference Update: HAVING: Stricter rules

UNION

Combines the results of two or more SELECT statements, which may involve rows from multiple tables or multiple sets from the same table, to produce a single result set (read-only), i.e. one dynamic table without duplicate rows. The unified columns in each separate output specification must match by degree (number and order of columns), type (data type) and size - what is known as union compatability. Which means they must each output the same number of columns in the same left-to-right order. Each column must also be consistent throughout in data type and size. By default UNION suppresses all duplicates in the final resulting sets. The ALL option keeps identical rows separate.

New to Firebird 2.0: Please refer to Enhancements to UNION handling for improvements of the rules for UNION queries.

See also:

Firebird 2.0 Language Reference Update: UNION

PLAN

Specifies the query plan, optionally included in the query statement, which should be used by the query optimizer instead of one it would normally choose.

```
<query_specification>
PLAN <plan_expr>

<plan_expr> =
   [JOIN | [SORT] [MERGE]] ({plan_item | plan_expr}
   [, {plan_item | plan_expr} ...])

<plan_item> = {table | alias}
```

```
   {NATURAL | INDEX (index [, index ...])| ORDER index}
```

where plan_item specifies a table and index method for a plan.

It tells the optimizer which indices, join order and access methods should be used for the query. Although the optimizer creates its own plan, and as a rule, usually selects the best method, there are situations where performance can be increased by specifying the plan yourself.

The IBExpert SQL Editor's Plan Analyzer and Performance Analysis allow the user to analyze and compare the optimizer's plan with their own.

Firebird 2.0's improvements to the PLAN clause can be referred to in the Firebird 2.0.4 Release Notes, Improvements in handling user-specified query plans.

See also:

SELECT

back to top of page

ORDER BY

The ORDER BY clause is used to sort a query's return sets, and can be used for any SELECT statement which is capable of retrieving multiple rows for output. It is placed after all other clauses (except a FOR UPDATE clause, if used, or a stored procedure's INTO clause).

The InterBase® 7.1 syntax is as follows:

```
order by <order_list>

where

<order_list> =
{col | int} [COLLATE collation]
[ASC[ENDING] | DESC[ENDING]]
[, order_list ...]
```

It specifies columns to order, either by column name or ordinal number in the query. Sorting items are usually columns. Ideal are indexed columns, as they are sorted much faster. A compound index may speed up performance considerable when sorting more than one column. N.B. Both columns and compound index need to be in an unbroken left-to-right sequence.

The comma-separated order_list specifies the order of the rows, complemented by ASCENDING (which is the default value, therefore it need not be explicitly specified) or DESCENDING or DESC.

If there is more than one sorting item, please note that the sorting precedence is from left to right.

The Firebird 1.5 syntax is slightly different:

```
ORDER BY <order_list>
   <list_item> = <column> | <expression> | <degree number>
```

```
                    ASC | DESC
                  [NULL LAST | NULLS FIRST]
```

Since Firebird 1.5 valid expressions are also allowed as sort items, even if the expression is not ouput as a runtime column. Sets can be sorted on internal or external function expressions or correlated subqueried scalars.

Firebird 1.5 supports the placement of NULLs, if and when present. The default value is NULLS LAST (sorts all nulls to the end of the return sets. NULLS FIRST needs to be explicitly specified, if null values are to be placed first.

New to Firebird 2.0: ORDER BY <ordinal-number> now causes SELECT * expansion - When columns are referred to by the ordinal number (degree) in an ORDER BY clause, when the output list uses SELECT * FROM ... syntax, the column list will be expanded and taken into account when determining which column the number refers to. This means that, now, SELECT T1.*, T2.COL FROM T1, T2 ORDER BY 2 sorts on the second column of table T1, while the previous versions sorted on T2.COL.

Tip: This change makes it possible to specify queries like SELECT * FROM TAB ORDER BY 5.

Firebird 2.0 also introduced some useful improvements to SQL sorting operations - please refer to Improvements in sorting in the Firebird 2.0.4. Release Notes and NULLs ordering changed to comply with standard in the Firebird 2.l Release Notes for details.

See also:

- Firebird 2.0 Language Reference Update: ORDER BY
- Firebird 2.0.4 Release Notes: Improvements in sorting

back to top of page

ROWS

```
ROWS value
   [TO upper_value]
   [BY step_value]
   [PERCENT][WITH TIES]
```

- value is the total number of rows to return if used by itself.
- value is the starting row number to return if used with TO.
- value is the percent if used with PERCENT.
- upper_value is the last row or highest percent to return.
- If step_value = n, returns every nth row, or n percent rows.
- PERCENT causes all previous ROWS values to be interpreted as percents.
- WITH TIES returns additional duplicate rows when the last value in the ordered sequence is the same as values in subsequent rows of the result set; must be used in conjunction with ORDER BY.

Please also refer to ROWS syntax for Firebird 2.0 syntax, description and examples.

See also:

- Firebird 2.0 Language Reference Update: ROWS
- Firebird Null Guide: Internal functions and directives

FOR UPDATE

```
[FOR UPDATE [OF col [, col ...]]
```

Only relevant when specifying columns listed after the SELECT clause of a DECLARE CURSOR statement that can be updated using a WHERE CURRENT OF clause.

Since Firebird 1.5 an optional WITH LOCK extension can be used with or without the FOR UPDATE syntax. Recommended however only for advanced developers as this supports a restricted level of explicit, row-level pessimistic locking.

RETURNING

The RETURNING clause syntax was implemented in Firebird 2.0 for the INSERT statement, enabling the return of a result set from the INSERT statement. The set contains the column values actually stored. Most common usage would be for retrieving the value of the primary key generated inside a BEFORE-trigger.

Available in DSQL and PSQL.

Syntax Pattern

```
INSERT INTO ... VALUES (...) [RETURNING <column_list> [INTO
<variable_list>]]
```

Example(s)

1.

```
INSERT INTO T1 (F1, F2)
   VALUES (:F1, :F2)
RETURNING F1, F2 INTO :V1, :V2;
```

2.

```
INSERT INTO T2 (F1, F2)
   VALUES (1, 2)
RETURNING ID INTO :PK;
```

Note:

1. The INTO part (i.e. the variable list) is allowed in PSQL only (to assign local variables) and rejected in DSQL.
2. In DSQL, values are being returned within the same protocol roundtrip as the INSERT itself is executed.
3. If the RETURNING clause is present, then the statement is described as

isc_info_sql_stmt_exec_procedure by the API (instead of isc_info_sql_stmt_insert), so the existing connectivity drivers should support this feature automatically.
4. Any explicit record change (update or delete) performed by AFTER-triggers is ignored by the RETURNING clause.
5. Cursor based inserts (INSERT INTO ... SELECT ... RETURNING ...) are not supported.
6. This clause can return table column values or arbitrary expressions.

Subquery functions

The functions ALL, ANY, SOME, EXISTS, and SINGULAR are available for subqueries. However it is not always necessary to use these, as the desired results can usually be attained by other means. However they can, in certain situations, improve performance.