Under the Hood : Data Access Paths

by Dimitry Yemanov, 2005

Contents

Introduction

This paper covers low-level query execution details that are the base for almost every DML statement. We'll describe various data source types as well as possible data tranformations. Also there will be some references to the optimizer logic (with chosen plan examples).

A set of data manipulations that the engine performs to return you the desired result set is called an access path. It may be represented with a tree which root node is a final result set. Every node of this tree is called an access method (or a data source). Data transformations performed within these access methods manipulate with data streams. Leaf nodes of the access path tree are called primary access methods, as their goal is to read the database file(s) and produce data sources for other access methods.

There are three kinds of data sources:

- primary access method - it reads records from tables or stored procedores
- filter method - it transforms one input stream to one output stream
- junction method - it transforms a number of input streams to a single output stream

Data sources could be pipelined or buffered. A pipelined date source returns rows as soon as it reads them from its input and peform the necessary transformation, i.e. rows are processed one by one. While a buffered data source have to read all rows from its input before it can return the first row.

From the performance point of view, every access method has two mandatory attributes: cardinality and cost. The former one reflects how many rows are expected to be read from a data source. The latter one estimates the cost of the access method execution. Cost is measured in logical reads (page fetches) that are necessary to return all rows from the given data source. Obviously, upper-level access methods include costs of the lower-level ones.

1. Primary access methods

This group of access methods creates a data stream based on the stored low-level objects, such as tables (internal and external) and procedures. Let's discuss each of these methods in details.

1.1. Direct table access

This primary access method is a basic one. It should be noted that we speak only about internal (stored) tables here. Access to external tables or stored procedures is performed differently and will be discussed later.

1.1.1. Full table scan

This method is also known as natural scan or sequential scan. The engine sequentially reads all pages allocated for

the given table in their physical order. Obviously, this method allows the highest thoughput, i.e. number of pages fetched per second. It's also obvious, that all table rows will be read regardless of whether we need some of them or not.

Rows are read in the pipeline mode and returned immediately one by one. Please note that there's neither prefetch nor buffering of rows (even within a page), so a full scan of 100K rows that consume 1000 data pages will be performed in 100K page fetches instead of 1000 (what could be supposed). Also, the engine doesn't support multi-block reads yet which could allow to fetch neighbour pages within a single batch, hence reducing page I/O.

The optimizer chooses a full table scan only if there are no indices that could be applied to the given search condition. Cost of this access method is equal to a number of rows in the table and it's estimated using number of allocated data pages, record length and average compression ratio on data pages.

Here and below sample SELECT queries along with their execution plans and schematic representations of the entire access path (in textual form) will be mentioned. A full table scan is marked as "NATURAL" in execution plans.

```
SELECT *
FROM RDB$RELATIONS

PLAN (RDB$RELATIONS NATURAL)

STATEMENT (SELECT)
[cardinality=500, cost=500.000]
  => TABLE (RDB$RELATIONS) SEQUENTIAL ACCESS
     [cardinality=500, cost=500.000]
```

The engine's execution statistics shows naturally scanned rows as non indexed reads.

1.1.2. Access via record identifier

The engine is given an identifier (physical number) of a record to read. This record identifier is known as a DB_KEY value at the DSQL level, but generally it's used by implementations of other access methods.

A physical record number contains information about a page number and an offset (slot) on this page. Both these values uniquely identify the given row and allow to fetch the required data page and find the record.

This access method is a low-level one and it's used only as an implementation of the bitmap-based scan and the index navigation. More about these access methods below.

Cost of the identifier-driven access is always equal to 1. The statistics shows the rows read via their identifiers as indexed reads.

1.1.3. Positioned access

We'll talk about positioned form of UPDATE and DELETE statements (syntax: WHERE CURRENT OF <cursor name>). There's an opinion that the positioned access is the same as select using RDB$DB_KEY, but this isn't actually true. Positioned access works only for an active cursor, i.e. for an already fetched row (via FOR SELECT or FETCH statements). Otherwise, an error isc_no_cur_rec (no current record for fetch operation) will be thrown. Therefore, positioned access is just a reference to a currently active row of the given cursor. Hence it doesn't require any read operations at all, while select via RDB$DB_KEY always require a data page fetch.

1.2. Index based access

The idea behind this access method is quite simple and well-known: aside from table data we also have a structure containing pairs "key - record number" in such a way to allow fast lookup of a given key value. In Firebird an index is implemented as a page B+ tree with a prefix key compression.

Indices could be simple (aka single-segment) or compound (aka multi-segment or composite). Please note that all fields of a composite index appear as a single key. An index could be looked up for an entire key as well as for its subkey (key substring). Obviously, lookup for a subkey is allowed for the starting part of a key only. If an index lookup is performed over all segments of a composite index, it's called a full key match, otherwise it's a partial key match. Considering a composite index over fields (A, B, C), the above means that:

- it could be used for booleans (A = 0) or (A = 0 and B = 0) or (A = 0 and B = 0 and C = 0), but it cannot be used for booleans (B = 0) or (C = 0) or (B = 0 and C = 0);
- the boolean (A = 0 and B > 0 and C = 0) will cause a partial key match over two segments, while the boolean (A > 0 and B = 0) will cause a partial key match over a single segment only.

In Firebird, index based access requires the engine to read index pages (to perform a lookup) as well as data pages (to read rows). Other DBMS's in some cases are able to perform an index-only scan, e.g. when all selected fields exist in the index. But this approach doesn't work in Firebird due to its MGA implementation, as every index key contains only a record identifier without information of its version, so the engine must read a data page anyway in order to find any version visible for the current transaction. It's often asked: what about including record versions (i.e. transaction numbers) into the index key to allow an index-only scan? There are two issues here. First, it would increase a key size, so an index would contain more pages, causing more I/O involved. Second, every record change will force index to be modified, even if none indexed fields were affected. While in the current

implementation, an index is modified only when indexed fields are changed.

If you were to compare various DBMS implementations, Firebird's indices have one more disadvantage - they're always scanned in one direction, from lower keys to upper ones, making some index operations (e.g. using an ascending index for a descending sort) impossible. The solution for this issue is being discussed by the developers. Note that such an issue doesn't affect index lookups (only index navigation is affected).

The major parameter that controls an optimization of the index based access is selectivity. This value reflects a number of unique keys in the given index. The optimizer assumes the uniformal distribution of index keys when making its decisions.

1.2.1. Bitmaps

The basic problem of the index access is a random I/O of data pages. Really, order of keys in the index almost never corresponds to order of rows in the table. So an attempt to read many rows via an index would cause multiple fetches of every data page. Other DBMS's solve this problem via clustering indices (MSSQL) or index-organized tables (Oracle), both of which store data in order of the index key or inside the index itself. Firebird has another solution for this issue. An index is scanned for the entire search range and all found matches (record numbers) are included into a special bitmap. This bitmap is implemented as a sparse array where every bit corresponds to a concrete record number to retrieve. By definition, this bitmap is ordered by record numbers, so appropriate data pages are fetched in their storage order. Bitmaps allow bitwise operators AND and OR, so the engine can use any number of indices for a retrieval.

As the bitmap-driven retrieval reads rows using identifier based access (which has cost = one), so the total cost would be a sum of costs for every index lookup involved plus cardinality of the bitmap.

1.2.2. Range index scan

Index lookup is performed using lower and upper bounds. It means that if a lower threshold exists in the search condition, then the engine initially finds the appropriate lower key and only then it starts to scan the keys and fill the bitmap. If the upper bound exists, then every scanned key is compared to the boundary value and the scan stops when the key becomes greater than the upper bound. This is called a range scan. If both bounds represent the same key, it's known to be an equality scan. If an equality scan involves a unique index, it's called a unique scan. The latter scan type have the special meaning as it can return no more than a single record and hence it has the least possible cost. If no bounds are specified, then it's a full index scan. This scan type is used for index navigation only (more below).

The known disadvantage of the Firebird's range scan implementation is the fact that both lower and upper bounds are strict. It means that for both search conditions (F > 0) and (F >= 0) zero will become a lower bound, causing the same number of indexed reads for both these conditions, although obviously some of them are redundant for the first sample. This issue is fixed in version 2.0.

It should be noted that starting with ODS 11.0 the engine skips NULL values while scanning the keys in the cases it's allowed. In many cases it causes better performance as the bitmap becomes smaller and hence less records are read. This option isn't used only for conditions IS [NOT] NULL and IS [NOT] DISTINCT FROM that must care about NULL values.

The optimizer applies a cost based strategy when choosing the indices for a retrieval. Cost of the range scan is estimated using index selectivity, number of keys, average key size and depth of the B+ tree. The range index scan is marked as "INDEX" in execution plans.

```
SELECT *
FROM RDB$INDICES
WHERE RDB$RELATION_NAME = ? AND RDB$FOREIGN_KEY = ?

PLAN (RDB$INDICES INDEX (RDB$INDEX_31, RDB$INDEX_41))

STATEMENT (SELECT)
[cardinality=2, cost=9.000]
  => TABLE (RDB$INDICES) ACCESS BY DB_KEY
     [cardinality=2, cost=9.000]
    => BITMAP AND
       [cardinality=2, cost=7.000]
      => INDEX (RDB$INDEX_31) RANGE SCAN
         [cardinality=5, cost=4.000]
      => INDEX (RDB$INDEX_41) RANGE SCAN
         [cardinality=2, cost=3.000]
```

1.2.3. Index navigation

This is also a sequantial scanning of index keys. The major difference between this method and the range scan (see above) is the lack of a bitmap between the index scan and identifier-driven table access, as we don't need sorting by record numbers this time. This means it's usually costy to use the index navigation as it's performed using not optimized (random) page I/O. So this access method is used with care for cases that could benefit from it.

Currently, there are two such cases. The first one is evaluation of aggregate functions MIN/MAX. It's easy to understand that it's enough to take the first (lower) key in the ASC index in order to evaluate MIN. If the fetched record isn't visible for our transaction, we take the next key and continue until we found the match. The second case is sorting or grouping of rows, as requested by ORDER BY or GROUP BY clauses. In this case we just walk the index and fetch all rows one by one. In both these cases records are read via their identifiers.

There are a few tricks that optimize this process. First, if there are search conditions that apply to the sort key, they create lower and/or upper bounds of the index scan. So in the case of (WHERE A > 0 ORDER BY A) the engine will perform a range scan instead of a full scan. Second, if there are other search conditions (that cannot be applied to the sort key) and they can be optimized via indices, the engine uses the combined algorithm where both bitmap and navigation are used. Let we have a query of type (WHERE A > 0 AND B > 0 ORDER BY A). In this case, the engine initially performs a range scan over the index of field "B" and creates the bitmap. Then value of "0" is used as a lower bound for a navigational scan over the index of field "A". Then the navigational scan starts and all scanned record numbers are checked against the bitmap. The data fetch is performed only if the bitmap contains this record number.

Cost of evalutating MIN/MAX is very small in most cases (a few page fetches). But in order to estimate a cost of the index-based sorting more information is required: number of keys and average key size, bitmap cardinality, clustering factor (how much storage of index keys corresponds to storage of their records).

Index navigation is marked as "ORDER" in execution plans. Starting with version 2.0, the plan may contain both "ORDER" and "INDEX" (if a bitmap scan is involved).

```
SELECT MIN(RDB$PROCEDURE_NAME)
FROM RDB$PROCEDURES
WHERE RDB$PROCEDURE_ID > ?

PLAN (RDB$PROCEDURES ORDER RDB$INDEX_21 INDEX (RDB$INDEX_22))

STATEMENT (SELECT)
[cardinality=100, cost=303.000]
  => TABLE (RDB$PROCEDURES) ACCESS BY DB_KEY
     [cardinality=100, cost=303.000]
    => INDEX (RDB$INDEX_21) FULL SCAN
       [cardinality=100, cost=203.000]
      => BITMAP
         [cardinality=100, cost=3.000]
        => INDEX (RDB$INDEX_22) RANGE SCAN
           [cardinality=100, cost=3.000]
```

1.3. Access using RDB$DB_KEY

Ways to work with RDB$DB_KEY are not documented. Those who're interested could find atricles of Claudio Valderrama: "The mystery of RDB$DB_KEY" and "Practical use of RDB$DB_KEY", that are placed on his site.

Implementation is pretty simple: the engine allocates a bitmap and puts a single record number (value of RDB$DB_KEY) there. Then the record is fetched via identifier-based access method. It's obvious that cost of using RDB$DB_KEY is equal to one. It looks like a pseudo index-based access and this is reflected in execution plans (an empty index name).

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$DB_KEY = ?

PLAN (RDB$RELATIONS INDEX ())

STATEMENT (SELECT)
[cardinality=1, cost=1.000]
  => TABLE (RDB$RELATIONS) ACCESS BY DB_KEY
     [cardinality=1, cost=1.000]
    => BITMAP
       [cardinality=1, cost=0.000]
      => DB_KEY
         [cardinality=1, cost=0.000]
```

Reads via RDB$DB_KEY are considered indexed ones. This is not formally correct, but it doesn't hurt either.

1.4. External tables

This access method is used to work with external tables only. In fact, it works quite similarly to a full table scan. Rows are read from the external file one by one via the current pointer value (offset). Any kind of caching is missing. External tables cannot be indexed.

As there are no alternative methods for external tables, its cost is neither used not even calculated. Its cardinality is considered equal to 10000. Execution plans show this method as "NATURAL".

```
SELECT *
FROM EXT_TABLE
WHERE EXT_FIELD = ?

PLAN (EXT_TABLE NATURAL)

STATEMENT (SELECT)
[cardinality=10000, cost=?.???]
  => TABLE (EXT_TABLE) SEQUENTIAL ACCESS
     [cardinality=10000, cost=?.???]
```

1.5. Procedures

This access method is used to select from stored procedures that issue the SUSPEND statement to return the result. In Firebird any stored procedure is considered to be a black box and its internals are never assumed by the engine. Procedure is a non-deterministic data source, so no indexing of its results is possible. Like the previously explained external tables access, procedural access also works similar to a full table scan. When a fetch from the procedure is performed, its code is executed from the moment of the previous stall point to the next SUSPEND statement, after that its output parameters create a new row to be returned.

Cost of procedural access isn't maintained either, as there are no alternatives. However, its cardinality is set to zero. Below we'll discuss this aspect and its side effects.

There are a few ways to display this access method in execution plans, depending on the engine's version. I consider the most correct way is to mark them as "NATURAL", although some versions may also embed the plans of the procedure internals in the "parent" plan.

```
SELECT *
FROM PROC_TABLE
WHERE PROC_FIELD = ?

PLAN (PROC_TABLE NATURAL)

STATEMENT (SELECT)
[cardinality=0, cost=?.???]
  => TABLE (PROC_TABLE) SEQUENTIAL ACCESS
     [cardinality=0, cost=?.???]
```

2. Filters

This group of methods tranforms the input data stream. Usually, they don't change the width of the input data, but decrease its cardinality accordingly to their rules. From the implementation point of view, all filters have one common thing - they don't have own cardinality and cost values. Instead, they inherit these values from the input data source.

2.1. Evaluation of booleans

This is a most commonly used type of a filter. It checks some search condition for every record passed. If this condition is evaluated to true then the record is returned untouched, otherwise it's ignored.

This filter implements clauses WHERE, HAVING and others. In order to decrease the total cardinality of the retrieval, evaluation of booleans is always performed as deep in the access path as possible. E.g. if you search for a specific value and then sort the results, the condition will be evaluated as soon as the row is fetched from its data page to avoid sorting redundant data.

Evaluation of booleans isn't reported in execution plans.

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME = ?

PLAN (RDB$RELATIONS INDEX (RDB$INDEX_0))

STATEMENT (SELECT)
[cardinality=1, cost=4.000]
  => BOOLEAN
     [cardinality=1, cost=4.000]
    => TABLE (RDB$RELATIONS) ACCESS BY DB_KEY
       [cardinality=1, cost=4.000]
      => BITMAP
         [cardinality=1, cost=3.000]
        => INDEX (RDB$INDEX_0) UNIQUE SCAN
           [cardinality=1, cost=3.000]
```

Looking at this example, someone could ask: why a BOOLEAN is used here if the search condition is implemented via a INDEX UNIQUE SCAN? The reason is that Firebird index lookups are not strict by design. In other words, a range index scan is just an optimization of the given condition which in some cases can return more rows than necessary. So the engine doesn't rely on index scans and always check the condition once more, after the row is fetched. This doesn't cause any noticable performance degradation in most cases.

2.2. Sorting

Also known as an external sort. This filter is used by the optimizer if there's a need to order the input stream and index navigation cannot be applied. Examples of its usage are: sorting or grouping, creation of an index B+ tree, evaluation of the DISTINCT clause, etc.

As the input stream is considered not ordered, the sorting filter have to read all the rows from its input stream before it can return any of them. Therefore, this filter implements a buffered data source.

The actual algorithm represents a multi-level quick sort. A number of input rows are placed inside the internal buffer, then it's sorted and moved to the external memory. After that, the next rows are processed similarly until the input stream reports EOF. Then the processed blocks are read and the binary merge tree is created. Fetching from the sorting filter means walking the tree and merging the rows. An external buffer could be the virtual memory as well as the disk space, depending on the firebird.conf settings.

Please note that records are read and sorted as a whole (not only sort keys). The engine never read the records from data pages again, they're returned from the sort buffer instead.

Sorting has two operational modes: normal and reducing. The former one keeps duplicate rows, while the latter removes them. For example, the DISTINCT clause is always processed in the reducing mode.

Sorting is reported as "SORT" in execution plans.

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME > ?
ORDER BY RDB$SYSTEM_FLAG

PLAN SORT (RDB$RELATIONS INDEX (RDB$INDEX_0))

STATEMENT (SELECT)
[cardinality=100, cost=105.000]
  => SORT
     [cardinality=100, cost=105.000]
    => BOOLEAN
       [cardinality=100, cost=105.000]
      => TABLE (RDB$RELATIONS) ACCESS BY DB_KEY
         [cardinality=100, cost=105.000]
        => BITMAP
           [cardinality=100, cost=5.000]
          => INDEX (RDB$INDEX_0) RANGE SCAN
             [cardinality=100, cost=5.000]
```

Any cases of redundant sorts (e.g. DISTINCT and ORDER BY clauses over the same field) are detected and removed by the optimizer.

2.3. Aggregation

This filter implements aggregate functions (MIN, MAX, AVG, SUM, COUNT), including cases of their usage in grouping.

Implementation is trivial. All these functions require a single register to store the current boundary (MIN/MAX) or accumulated (others) value. For each input row the engine checks or increments the register. But when grouping is in a game, algorithm becomes more complex. In order to evaluate the aggregates correctly, every group should be clearly distinguished from others. The easiest solution is to guarantee that grouping keys are ordered in advance, and this approach is used in Firebird - aggregation is always performed on the sorted values.

There's also an alternative way to evaluate aggregates - hashing. In this case, ordering of the input stream isn't important. Instead, a hash function is applied to every grouping key and the registers are stored on a per key basis inside a hash table. An obvious advantage of this algorithm - there's no extra sort. A disadvantage - a hash table requires more memory. This method usually wins when the grouping key has quite low selectivity (a few unique values, hence a hash table is small). There's no hashing aggregation in Firebird, although it could be considered for implementation.

Aggregation isn't reported in execution plans.

```
SELECT RDB$SYSTEM_FLAG, COUNT(*)
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME > ?
GROUP BY RDB$SYSTEM_FLAG
```

```
PLAN SORT (RDB$RELATIONS INDEX (RDB$INDEX_0))

STATEMENT (SELECT)
[cardinality=100, cost=105.000]
  => AGGREGATE
     [cardinality=100, cost=105.000]
    => SORT
       [cardinality=100, cost=105.000]
      => BOOLEAN
         [cardinality=100, cost=105.000]
        => TABLE (RDB$RELATIONS) ACCESS BY DB_KEY
           [cardinality=100, cost=105.000]
          => BITMAP
             [cardinality=100, cost=5.000]
            => INDEX (RDB$INDEX_0) RANGE SCAN
               [cardinality=100, cost=5.000]
```

Note than when functions SUM/AVG/COUNT are evaluated in the DISTINCT mode, the engine performs a reducing sort for every group of input values. But there's no "SORT" mark in execution plans in this case.

2.4. Counters

These filters is also very simple. Their goal is to return only part of the input stream, based on some value N of the internal counter. There are two kinds of this filter, that are used to implement clauses FIRST/SKIP/ROWS. The first one is a FIRST-counter and it returns only the first N rows from its input stream and then reportes EOF. The second one is a SKIP-counter which ignores the first N input rows and then returns others starting with N+1.

Counters are not reported in execution plans .

```
SELECT FIRST 1 SKIP 1 *
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME > ?

PLAN (RDB$RELATIONS INDEX (RDB$INDEX_0))

STATEMENT (SELECT)
[cardinality=100, cost=105.000]
  => FIRST
     [cardinality=100, cost=105.000]
    => SKIP
       [cardinality=100, cost=105.000]
      => BOOLEAN
         [cardinality=100, cost=105.000]
        => TABLE (RDB$RELATIONS) ACCESS BY DB_KEY
           [cardinality=100, cost=105.000]
          => BITMAP
             [cardinality=100, cost=5.000]
            => INDEX (RDB$INDEX_0) RANGE SCAN
               [cardinality=100, cost=5.000]
```

2.5. Singularity check

This filter is targeted to guarantee that only a single record exists in its input stream. It's applied when singleton subqueries are used.

This filter performs two fetches from the input stream. If the second one returned EOF, the first record is returned. Otherwise, an error isc_sing_select_err (multiple rows in singleton select) is thrown.

Singularity check is not reported in execution plans.

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$RELATION_ID = ( SELECT RDB$RELATION_ID FROM RDB$DATABASE )

PLAN (RDB$DATABASE NATURAL)
PLAN (RDB$RELATIONS INDEX (RDB$INDEX_1))

STATEMENT (SELECT)
[cardinality=1, cost=1.000]
  => SINGULAR
     [cardinality=1, cost=1.000]
    => TABLE (RDB$DATABASE) SEQUENTIAL ACCESS
       [cardinality=1, cost=1.000]
```

```
STATEMENT (SELECT)
[cardinality=1, cost=4.000]
  => BOOLEAN
     [cardinality=1, cost=4.000]
    => TABLE (RDB$RELATIONS) ACCESS BY DB_KEY
       [cardinality=1, cost=4.000]
      => BITMAP
         [cardinality=1, cost=3.000]
        => INDEX (RDB$INDEX_1) RANGE SCAN
           [cardinality=1, cost=3.000]
```

2.6. Record locking

This filter has been added in Firebird 1.5 and it implements pessimistic row locking. It's used only when a WITH LOCK clause is specified for the retrieval. Every record that's read from the input stream is locked and then returned. The locking is implemented via a dummy record version, marked with a current transaction identifier.

Current versions allow record locking only for primary access methods. Locking is not reported in execution plans.

```
SELECT *
FROM RDB$RELATIONS
WITH LOCK

PLAN (RDB$RELATIONS NATURAL)

STATEMENT (SELECT)
[cardinality=100, cost=100.000]
  => LOCK
     [cardinality=100, cost=100.000]
    => TABLE (RDB$RELATIONS) SEQUENTIAL ACCESS
       [cardinality=100, cost=100.000]
```

3. Junctions

This group of access methods behaves very similar to filters. The formal difference is that junctions always operate with a few input data streams. Also, usually they extend the width of the stream or increase its cardinality.

There are two junction classes - joins and unions, targeted at different goals. Let's see why they are needed and how they could be implemented.

3.1. Joins

As it can be concluded from its name, these access methods implement SQL joins. The SQL specification declares two kinds of joins: inner and outer. Besides, outer joins could be one-way (left or right) and full. Every join has two input streams - left and right ones. For an inner join and a full outer join these streams are semantically equivalent. For the one-way outer join one of the streams is called a master and the second one is a slave. Also the master stream is often refered to as an outer one and the slave stream as an inner one. For a left outer join the master (outer) stream is the left one, while the slave (inner) stream is the right one. For a right outer join the situation is opposite, as formally a right outer join is a reversed left outer join.

Every join also has a link condition which defines the result of a join, i.e. how rows of both input streams correspond to each other. If there's no link condition, we have a border case - full production (cross join) of input stream.

Let's get back to one-way outer joins. There's an important reasoning behind why their streams are called master and slave. In this case an outer (master) stream have to be read before an inner (slave) one, otherwise it would be impossible to return the correct result. It could be concluded that the optimizer has no control over an order of streams evaluation and such an order is always dictated by the query source. For other join types, streams are independent and an order of their evaluation is determined by the optimizer.

3.1.1. Nested iteration

This join algorithm is very often used in Firebird. It's also known as a nested loops join or a recursive join.

It works the following way. Initially, an outer stream is opened and one row is fetched. After that, an inner stream is opened and also fetched once. Then the engine evaluates the link condition. If this condition is satsfied, both fetched records are combined and the result is returned. Then this iteration continues for every inner row until EOF is encountered. Then we do the same for the next row of the outer stream and so on.

The key point of this algorithm is a nested (recursive) retrieval from the inner stream. Obviously, it's very costly to read the entire inner stream for every outer row, so this method is effective only if an index could be applied to a link condition in order to limit the number of rows fetched for every outer iteration. As both AND and OR conditions can be optimized via bitmaps, the nested iteration can be effectively used for almost all kinds of join conditions.

The recursive join implementation is chosen by the optimizer always when there are indices that could be used

there. Its reported as "JOIN" in execution plans.

```
SELECT *
FROM RDB$RELATIONS R
JOIN RDB$RELATION_FIELDS RF ON R.RDB$RELATION_NAME = RF.RDB$RELATION_NAME
JOIN RDB$FIELDS F ON RF.RDB$BASE_FIELD = F.RDB$FIELD_NAME

PLAN JOIN (RF NATURAL, F INDEX (RDB$INDEX_2), R INDEX (RDB$INDEX_0))

STATEMENT (SELECT)
[cardinality=2500, cost=17500.000]
  => LOOP (INNER)
     [cardinality=2500, cost=17500.000]
    => TABLE (RDB$RELATION_FIELDS) SEQUENTIAL ACCESS
       [cardinality=2500, cost=2500.000]
    => TABLE (RDB$FIELDS) ACCESS BY DB_KEY
       [cardinality=1, cost=3.000]
      => BITMAP
         [cardinality=1, cost=2.000]
        => INDEX (RDB$INDEX_4) RANGE SCAN
           [cardinality=1, cost=2.000]
    => TABLE (RDB$RELATIONS) ACCESS BY DB_KEY
       [cardinality=1, cost=3.000]
      => BITMAP
         [cardinality=1, cost=2.000]
        => INDEX (RDB$INDEX_0) UNIQUE SCAN
           [cardinality=1, cost=2.000]
```

The recursive algorithm has one known pitfall - it never uses indices for a full outer join. Of course, this makes such a join quite slow. By the way, it's very easy to find problematic spots in the nested loops retrieval: if any of the joined streams (except the first one) is performed via "NATURAL", it always means slow execution.

Now let's get back to stored procedures. As it has been said before, cardinality of procedural access is always zero. What does it cause? It automatically places the procedure in the beginning of an inner join. Therefore, it will be executed only one time instead of being executed again and again for every outer iteration (as indices cannot be applied here). However, there's also an issue here. The optimizer doesn't check input parameters that can possibly depend on other streams:

```
SELECT *
FROM MY_TAB, MY_PROC(MY_TAB.F)
```

or

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

In this case the procedure will be executed first, when there are no rows fetched from table MY_TAB yet. Accordingly, a runtime error isc_no_cur_rec (no current record for fetch operation) will be thrown. This issue can be worked around using an outer join instead:

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

Now the procedure will be always executed after the table has been fetched.

3.1.2. One-way merge

A merge is an alternative algorithm to perform a join. Input streams are independent in this case, but they're required to be ordered by the link key. The engine makes a binary tree of the merge and then rows are fetched from both streams and their keys are compared for equality. If the keys are the same, the result of a join is return. Then new rows are read and the process continues. The difference from the previously explained method is that a merge is performed in a single run, so every input stream is read only once.

However, a merge cannot be used for every join variant. As it has been mentioned earlier, only equality based joins are allowed. Therefore, joins like (ON MASTER.F > SLAVE.F) cannot be processed by this algorithm. But this method has one advantage over the recursive algorithm - it allows effective execution of expression based joins (e.g. ON MASTER.F + 1 = SLAVE.F + 2). A merge algorithm is able to process complex AND-combined join conditions (input streams are just sorted by a number of fields), although OR-combined join conditions cannot be processed.

It should also be noted that neither of the outer join types can be performed using a merge. This limitation is going to be addressed in future versions.

The optimizer chooses a merge join only when a recursive join cannot be done at all or it would be not efficient. Mostly this covers cases when there are no indices or they cannot be applied, or also when there are no

dependencies between input streams. A merge algorithm is reported as "MERGE" in execution plans.

```
SELECT *
FROM RDB$RELATIONS R
JOIN RDB$RELATION_FIELDS RF ON UPPER(R.RDB$RELATION_NAME) = UPPER(RF.RDB$RELATION_NAME)
WHERE R.RDB$SYSTEM_FLAG = 1

PLAN MERGE (SORT (RF NATURAL), SORT (R NATURAL))

STATEMENT (SELECT)
[cardinality=2500, cost=3000.000]
  => MERGE
     [cardinality=2500, cost=3000.000]
    => SORT
       [cardinality=500, cost=500.000]
      => BOOLEAN
         [cardinality=500, cost=500.000]
        => TABLE (RDB$RELATIONS) SEQUENTIAL ACCESS
           [cardinality=500, cost=500.000]
    => SORT
       [cardinality=2500, cost=2500.000]
      => TABLE (RDB$RELATION_FIELDS) SEQUENTIAL ACCESS
         [cardinality=2500, cost=2500.000]
```

3.1.3. Hashing

This is yet another alternative join method. In this case, input streams are always separated to master and slave ones. Initially, the slave stream is entirely read into the internal buffer. While reading, a hash function is applied to every join key and a pair {hash, pointer} is written to the hash table. Then the master stream is being read and the join key of every master record is probed against the hash table. If there's a match, records are joined and returned, otherwise we continue scanning the master stream.

It's obvious, that a hash join has the same advantages and pitfalls as a merge join: indices are not used for a join, expressions are allowed, only equality based joins can be processed.

This algorithm is mostly effective for a small hash table, so the smallest stream is chosen to become the slave one. In this case, a hash join always win over a merge join which requires an extra sort of input streams. However, when both streams are large or when the bigger stream is chosen as a slave one (consider outer joins), this method performs slower than others.

There's no hash join implementation in the current Firebird versions.

3.2. Unions

The name speaks for itself. This access method is responsible for a SQL union operation. This operation has two modes: ALL and DISTINCT. The former one implementation is trivial: the engine reads the first input stream and returns every its row until EOF is encountered, then the second input stream is returned and so on. The DISTINCT mode requires to remove full duplicates of processed rows. In roder to achieve this, a reducing sorting filter is placed at the output of the union source.

Both cardinality and cost of this method are calculated as a sum of these values for all input streams. An union itself isn't reported in execution plans, but independent plans of its input streams are reported instead.

```
SELECT RDB$RELATION_ID
FROM RDB$RELATIONS
WHERE RDB$SYSTEM_FLAG = 1
UNION
SELECT RDB$PROCEDURE_ID
FROM RDB$PROCEDURES
WHERE RDB$SYSTEM_FLAG = 1

PLAN (RDB$RELATIONS NATURAL)
PLAN (RDB$PROCEDURES NATURAL)

STATEMENT (SELECT)
[cardinality=1500, cost=1500.000]
  => SORT
     [cardinality=1500, cost=1500.000]
    => UNION
       [cardinality=1500, cost=1500.000]
      => BOOLEAN
         [cardinality=500, cost=500.000]
        => TABLE (RDB$RELATIONS) SEQUENTIAL ACCESS
           [cardinality=500, cost=500.000]
      => BOOLEAN
```

```
 [cardinality=1000, cost=1000.000]
=> TABLE (RDB$PROCEDURES) SEQUENTIAL ACCESS
   [cardinality=1000, cost=1000.000]
```

Conclusion

We've just reviewed all operations implementing any kind of SQL retrieval. You've seen description and examples for every algorithm, incuding the detailed information about access paths items. It could be mentioned that Firebird implements not so many access methods as being compared with its competitors, however it's usually explained by either architectural reasons or more effective existing algorithms.

It's important to understand that most of the operations are actually performed during the fetch instead of being executed and entirely cached at the beginning. It usually causes quite fast response and less memory usage. Exceptions from this rule are an external sort and dependent operations (e.g. GROUP BY or UNION DISTINCT).