Data access methods used in Firebird

Simonov Denis, Yemanov Dmitry

Version 2.0 by 2025-03-27

Table of Contents

This material is sponsored and created with the sponsorship and support of IBSurgeon https://www.ib-aid.com, vendor of HQbird (advanced distribution of Firebird) and supplier of performance optimization, migration and technical support services for Firebird.

Preface

This article describes the access methods existing in **Firebird 5.0**. It is based on the original article Firebird: data access methods by Dmitry Yemanov. The previous article was written a long time ago and is relevant for Firebird 2.0 and partially for newer versions of Firebird.

The following has been changed compared to the original article:

- Added descriptions of access methods that appeared after Firebird 2.0 — in 2.1, 2.5, 3.0, 4.0 and Firebird 5.0.
- Replaced the schematic query execution tree with real Explain plans (appeared in Firebird 3.0).
- Since the original article provided examples of calculating cardinality and cost, they were added to the Explain plans. Cardinality values were obtained using the `RDB$SQL.EXPLAIN` procedure from Firebird 6.0. Costs were calculated using formulas that were either in the original article or were obtained by me while analyzing the Firebird source code.
- Added a formula for calculating the cost of Hash Join.
- Added a description of how filters cut cardinality (Filter, group by, distinct).

Chapter 1. Terminology

Data access path is a set of operations on data performed by the server to obtain the result of a given selection. Explain plan is a tree with the root representing the final result. Each node of this tree is called **data access method** or **data source**. The objects of operations in access methods are **data streams**. Each access method either forms a data stream or transforms it according to certain rules. Leaf nodes of the tree are called **primary data access methods**. Their only task is to form data streams.

In terms of the types of operations performed, there are three classes of data sources:

- primary data access methods — reads from a table or stored procedure, forms the initial data stream;
- filters — transforms one input data stream into one output stream;
- merges — transforms two or more input data streams into one output stream.

Data sources can be pipelined or buffered. A pipelined data source produces records as it reads its input streams, while a buffered source must first read all the records from its input streams before it can produce the first record on its output.

In terms of performance evaluation, each access method has two mandatory attributes — cardinality and cost. The first reflects how many records will be selected from the data source. The second estimates the cost of executing the access method. The cost directly depends on the cardinality and the mechanism of data flow selection or transformation. In current server versions, the cost is determined by the number of logical reads (page fetches) required to return all records by the access method. Thus, "higher" methods always have a higher cost than low-level ones. The CPU load in computational access methods is reduced to an approximately equivalent (by specified coefficients) number of logical reads.

Chapter 2. Primary data access methods

This group of access methods performs data flow creation based on low-level sources such as tables (internal and external) and procedures. We will consider each of the primary data sources separately below.

2.1. Reading the table

Is the most common primary access method. It is worth noting that here we are talking only about “internal” tables, that is, those whose data is located in database files. Access to external tables (external tables), as well as selection from stored procedures (stored procedures) are carried out in other ways and will be described separately.

2.1.1. Table Full Scan

This method is also known as Full Scan, Natural Scan, Sequential Scan.

In this access method, the server performs a sequential reading of all pages allocated for a given table, in the order of their physical location on the disk. Obviously, this method provides the highest performance in terms of throughput, i.e. the number of records fetched per unit of time. It is also quite obvious that all records of a given table will be read, regardless of whether we need them or not.

Since the expected data volume is quite large, there is a problem of the read pages displacing other pages that are potentially needed by competing sessions during the process of reading the table pages from disk. For this purpose, the logic of the page cache operation changes — the current scan page is located in the MRU (most recently used) position during the reading of all records from this page. As soon as there is no more data on the page and the next one needs to be fetched, the current page is released with the LRU (least recently used) flag, going to the "tail" of the queue and thus being the first candidate for removal from the cache.

The records are read from the table one by one, immediately issued to the output. It should be noted that there is no pre-fetch of records (at least within the page) with their buffering in the server. That is, a full selection from a table with 100K records occupying 1000 pages will lead to 100K page fetches, and not to 1000, as one might expect. There is also no batch reading (multi-block reads), in which adjacent pages could be allocated into groups and read from the disk "in batches" of several pieces, thereby reducing the number of physical I/O operations.

Both of these features are planned for implementation in future versions of Firebird.

Starting with Firebird 3.0, Data Pages (DPs) for tables (containing more than 8 DPs) are allocated in groups called **extents**. One extent consists of 8 pages. This allows the engine to use the OS prefetch more efficiently, since the pages related to one table are located close to each other, and therefore a full table scan is more likely to read the required pages from the OS cache.

The optimizer uses a rule-based strategy when choosing this access method — a full scan is

performed only if there are no indexes applicable to the query predicate. The cost of this access method is equal to the number of records in the table (estimated approximately by the number of table pages, the record size, and the average compression ratio of records on data pages). Theoretically, this is incorrect and the choice of access method should always be based on cost, but in practice this turns out to be unnecessary in the vast majority of cases. The reasons for this will be explained below when describing index access.

Here and below, when mentioning the optimizer behavior, the following will be given: an example of a SELECT query, its execution plan in Legacy and Explain forms. In the Legacy execution plan, a full table scan is designated by the word “NATURAL”. In the Explain form — `Table "<table name>" Full Scan`. Currently, what is specified in square brackets (cardinality and cost) is not displayed in the explain plan, it is given as an example of calculating these values.

Example 1. Full scan of RDB$RELATIONS table

```
SELECT *
FROM RDB$RELATIONS
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=120.0, cost=120.0]
Select Expression
    [cardinality=120.0, cost=120.0]
    -> Table "RDB$RELATIONS" Full Scan
```

In the query execution statistics, records read by a full scan are reported as non indexed reads.

If you use table aliases, the plan will also show the aliases.

Example 2. Full table scan RDB$RELATIONS with alias

```
SELECT *
FROM RDB$RELATIONS R
```

```
PLAN (R NATURAL)
```

```
[cardinality=120.0, cost=120.0]
Select Expression
    [cardinality=120.0, cost=120.0]
    -> Table "RDB$RELATIONS" as "R" Full Scan
```

2.1.2. Data access via record ID

In this case, the execution engine obtains the identifier (physical number) of the record to be read. This record number can be obtained from various sources, each of which is described below.

In simplified terms, the physical record number contains information about the page on which the record is located and the offset within that page. This information is therefore sufficient to fetch the required page and find the desired record on it.

This access method is low-level and is used only as an implementation of bitmap-based fetching (both index scan and `RDB$DB_KEY` access) and index navigation. More details on these access methods will be given later.

The cost of this type of access is always equal to one. Reads of records are reflected in statistics as indexed.

2.1.3. Positioned data access

Here we are talking about positioned statements `UPDATE` and `DELETE` (syntax `WHERE CURRENT OF`). There is a misconception that this access method is syntactically analogous to fetching via `RDB$DB_KEY`, but this is not true. Positioned data access works only for the active cursor, i.e. for the already fetched record (using `FOR SELECT` or `FETCH` statements). Otherwise, the error `isc_no_cur_rec` ("no current record for fetch operation") will be thrown. Thus, this is simply a way to refer to the active record of the cursor, which does not require any read operations at all. Whereas fetching via `RDB$DB_KEY` involves access via the record identifier and, therefore, always results in fetching one page.

2.2. Indexed data access

The idea of index access to data is simple — in addition to the table with data, we also have a structure containing "key - record number" pairs in a form that allows for fast search by key value. In Firebird, the index is a page B+ tree with prefix key compression.

Indexes can be simple (single-segment) and composite (multi-segment). It should be noted that the set of fields of a composite index is a single key. Search in the index can be performed either by the entire key or by its substring (subkey). Obviously, search by subkey is allowed only for the initial part of the key (for example, `STARTING WITH` or using not all segments of the composite). If the search is performed by all segments of the index, then this is called a full match of the key, otherwise it is a partial match of the key. From this it follows that for a composite index by fields (A, B, C):

- it can be used for predicates (`A = 0`) or (`A = 0 and B = 0`) or (`A = 0 and B = 0 and C = 0`), but it cannot be used for predicates (`B = 0`) or (`C = 0`) or (`B = 0 and C = 0`);
- the predicate (`A = 0 and B > 0 and C = 0`) will result in a partial match on two segments, and the predicate (`A > 0 and B = 0`) will result in a partial match on only one segment.

Obviously, index access requires us to read both index pages for searching and data pages for reading records. Other DBMSs may be able to limit themselves to index pages only in some cases, for example if all fields in a selection are indexed. But this scheme is impossible in Firebird due to its architecture — the index key contains only the record number without any version information — so the server must read the record itself anyway to determine if any version with

the given key is visible to the current transaction. A question often arises: what if we include version information (i.e. transaction numbers) in the index key? Then we can implement pure index coverage (index only scan). But there are two problematic points here. First, the key length will increase, so the index will occupy more pages, which will lead to more I/O for the same scan operation. Second, every change to a record will lead to a modification of the index key, even if non-key fields were changed. While now the index is modified only if the key fields of a record are changed. The first problem will lead to a significant deterioration in index scanning performance with short keys (INT, BIGINT, and other types). The second problem at least triples the number of modified pages per data modification command compared to the current situation. Until now, server developers consider this too high a price to pay for implementing pure index coverage.

There is an alternative to pure index scanning implemented in Postgres. It works effectively for tables in which only a small portion of the records are changed. For this purpose, in addition to the index itself, there is an additional disk structure called the visibility map. The index scan procedure, having found a potentially suitable record in the index, checks the bit in the visibility map for the corresponding data page. If it is set, then the row is visible, and the data can be returned immediately. Otherwise, it will have to visit the row record and check whether it is visible, so there will be no gain compared to a regular index scan.

This option can be implemented in Firebird as well. Starting with Firebird 3.0, there is a so-called swept flag for DP (Data Page) pages. It is set to 1 if a sweep or garbage collector visited the page and did not find or cleared all non-primary versions of records. The same flag is present on PP (Pointer Page) pointer pages, which can be used as an analogue of the visibility map described above.

Compared to other DBMSs, Firebird indexes have another feature: index scanning is always one-way, from smaller keys to larger ones. Often, because of this, an index is called one-way, and its node is said to have pointers only to the next node and no pointer to the previous one. In fact, this is not the problem. All Firebird server disk structures are designed as deadlock free, and the minimum possible granularity of locks is guaranteed. In addition, there is also a rule of "careful" writing of pages, which serves for instant recovery after a failure. The problem with bidirectional indexes is that they violate this rule when splitting a page. At the moment, there is no known way to work with bidirectional pointers without locks in the case when one of the pages must be written strictly before the other.

In fact, there are options to work around this problem and they are currently being discussed by developers.

This feature makes it impossible to use an ASC index for DESC sorting or `MAX` calculations, and vice versa, it makes it impossible to use a DESC index for ASC sorting or `MIN` calculations. Of course, unidirectionality does not interfere with index scanning for search purposes.

2.2.1. Index selectivity

The main parameter that influences index access optimization is the index **selectivity**. This is the reciprocal of the number of unique key values. Cardinality and cost calculations assume a uniform distribution of key values in the index.

The selectivity of an index can be found using the following query

```
SELECT RDB$STATISTICS
FROM RDB$INDICES
WHERE RDB$INDEX_NAME = '<index_name>'
```

Example 3. Retrieving saved statistics for the CUSTNAMEX index

```
SELECT RDB$STATISTICS
FROM RDB$INDICES
WHERE RDB$INDEX_NAME = 'CUSTNAMEX'
```

```
         RDB$STATISTICS
=======================
    0.06666667014360428
```

For composite indexes, in addition to the index selectivity, Firebird also stores the selectivity of the set of its segments, starting from the first to the given one. The selectivity of the set of segments can be found using a query to the RDB$INDEX_SEGMENTS table.

Example 4. Retrieving saved statistics for each segment of the NAMEX index

```
SELECT RDB$FIELD_NAME, RDB$FIELD_POSITION, RDB$STATISTICS
FROM RDB$INDEX_SEGMENTS
WHERE RDB$INDEX_NAME = 'NAMEX';
```

```
RDB$FIELD_NAME     RDB$FIELD_POSITION          RDB$STATISTICS
================== ================== =======================
LAST_NAME                           0     0.02500000037252903
FIRST_NAME                          1     0.02380952425301075
```

Example 5. Retrieving saved statistics for the NAMEX index

```
SELECT RDB$STATISTICS
FROM RDB$INDICES
WHERE RDB$INDEX_NAME = 'NAMEX'
```

```
         RDB$STATISTICS
=======================
    0.02380952425301075
```

The index selectivity is calculated when it is built (`CREATE INDEX`, `ALTER INDEX ⋯ ACTIVE`) and may become outdated later. To update the index statistics, use the following SQL query

```
SET STATISTICS INDEX <index_name>;
```

Example 6. Collecting statistics for the NAMEX index

```
SET STATISTICS INDEX NAMEX;
```

Currently, Firebird does not update index statistics automatically. In addition, the stored statistics contain very little information for optimizing access using the index. In fact, only the selectivity of the index and its segments is stored. But such important index characteristics as NULL value selectivity, index depth, average index key length, histograms of value distribution, and clustering factor are not stored.

The stored statistics will be expanded in the next versions of Firebird. In addition, some features for its automatic updating are planned.

In addition to simple and composite indexes, Firebird allows you to create indexes by expression. In this case, instead of table field values, the index keys are the values of an expression that returns a scalar value. To use such indexes, the optimizer must use the same expression in the query filter condition as was specified when creating the index. Indexes by expression cannot be composite, but the expression can contain several table fields. The selectivity of such an index is calculated in the same way as for regular indexes.

2.2.2. Partial indices

Starting with Firebird 5.0, when creating an index, it is possible to specify an optional WHERE clause that defines a search condition that limits the subset of table records to be indexed. Such indexes are called partial indexes. The search condition must contain one or more table columns.

The optimizer can use a partial index only in the following cases:

- the `WHERE` condition includes exactly the same logical expression as defined for the index;
- the search condition defined for the index contains logical expressions combined by OR, and one of them is explicitly included in the `WHERE` condition;
- the search condition defined for the index specifies `IS NOT NULL`, and the `WHERE` condition includes an expression for the same field that is known to ignore `NULL`.

I will demonstrate the last point with an example. Suppose you have created the following partial index:

```
CREATE INDEX IDX_HORSE_DEATHDATE
ON HORSE(DEATHDATE) WHERE DEATHDATE IS NOT NULL;
```

Now the optimizer can use this index not only for the `IS NOT NULL` predicate, but also for the =, <, >, `BETWEEN` and other predicates, if they do not return `TRUE` when compared with `NULL`. For the `IS NULL` and `IS NOT DISTINCT FROM` predicates, the index cannot be used.

```
-- partial index can be used
SELECT *
FROM HORSE
WHERE DEATHDATE IS NOT NULL

-- partial index can be used
SELECT *
FROM HORSE
WHERE DEATHDATE = ?

-- partial index can be used
SELECT *
FROM HORSE
WHERE DEATHDATE BETWEEN ? AND ?

-- partial index cannot be used
SELECT *
FROM HORSE
WHERE DEATHDATE IS NOT DISTINCT FROM ?
```

If a regular index and a partial index exist on the same set of fields, the optimizer will choose the regular index in most cases, even if the `WHERE` clause includes the same expression as defined in the partial index. The reason for this behavior is that the selectivity of a regular index is known "exactly" (calculated as the inverse of the number of unique keys). But fewer keys are included in a partial index due to the additional filtering condition, so the stored selectivity of the index will be worse. The calculated selectivity of a partial index is estimated as the stored selectivity multiplied by the proportion of records included in the index. This proportion is currently not stored in statistics ("exactly" is not known), but is estimated based on the selectivity of the filter expression specified when creating the index (see Predicate checking). However, the actual selectivity may be lower, and therefore a regular index (with an exactly calculated selectivity value) may win.

This may change in future versions of Firebird.

2.2.3. Bitmaps

The main standard problem with index access is random I/O to data pages. Indeed, the order of keys in the index very rarely coincides with the order of the corresponding records in the table. Thus, selecting a significant number of records through the index will most likely lead to multiple fetching of each corresponding data page. This problem is solved in other DBMSs using clustered indexes (MSSQL term) or index-ordered tables (Oracle term), in which the data is located directly in the index in ascending order of the cluster key.

Firebird solves this problem in a different way. Index scanning is not a pipeline operation, but is performed for the entire search range, including the obtained record numbers in a special bitmap. This map is a sparse bit array, where each bit corresponds to a specific record, and the presence of one in it is an indication for selecting this record. The peculiarity of this solution is that the bitmap

is sorted by record numbers by definition. After the scan is complete, this array serves as the basis for sequential access via the record identifier. The advantage is obvious - reading from the table is done in the physical order of the pages, as with a full scan, i.e. each page will be read no more than once. Thus, the simplicity of the index implementation is combined with the most efficient access to records. The price is a slight increase in response time for queries with the `FIRST ROWS` strategy, when you need to get the first records very quickly.

Bitmaps allow intersection (bit and) and union (bit or) operations, so the server can use multiple indexes on a single table.

2.2.4. Range Scan

The search under the index is performed using the upper and lower bounds. That is, if the lower scan bound is specified, the corresponding key is found first and only then the sequential search of keys begins with the entry numbers in the bitmap. If the upper bound is specified, each key is compared with it and when it is reached, the scan stops. This mechanism is called range scan. In the case where the lower bound key is equal to the upper bound key, we speak of an equality scan. If the equality scan is performed for a unique index, this is called a unique scan. This type of scan is of particular importance, since it can return no more than one entry and, therefore, is by definition the cheapest. If none of the bounds are specified, we are dealing with a full scan. This method is used exclusively for index navigation, described below.

When possible, the server skips NULL values when scanning a range, if it is acceptable to do so. In most cases, this results in improved performance due to a smaller bitmap size and, therefore, fewer indexed reads. This option is not used only for indexed predicates such as `IS NULL` and `IS NOT DISTINCT FROM`, which are NULL aware.

When choosing indexes to scan, the optimizer uses a cost-based strategy. The cost of a range scan is estimated based on the index selectivity, the number of records in the table, the average number of keys per index page, and the height of the B+ tree.

The height of the B+ tree is currently not stored in statistics, and is therefore specified in the code as a constant equal to 3.

In a Legacy execution plan, an index range scan is denoted by the word “INDEX” followed in parentheses by the names of all the indexes that make up the resulting bitmap.

In the Explain plan, the display of the range scan is somewhat more complex. In general, it looks like this:

```
Table "<table_name>" Access By ID
    -> Bitmap
        -> Index "<index_name>" <scan_type>
```

Where `index_name` is the name of the index to use, `scan_type` is the scan type, `table_name` is the table name.

Unique scan

A unique scan can only be used for unique indexes when scanning across all segments (exact match) and using only equality predicates "=". Unique indexes are automatically created for primary key constraints or unique constraints, but can also be created manually. This type of scan is of particular importance, since it can return no more than one record, and therefore its cardinality is always equal to 1. The cost of this is calculated using the formula

```
cost = indexDepth + 1
```

Example 7. Index Unique Scan

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

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

In the Explain plan, a unique scan is displayed as follows:

```
[cardinality=1.0, cost=4.000]
Select Expression
    [cardinality=1.0, cost=4.000]
    -> Filter
        [cardinality=1.0, cost=4.000]
        -> Table "RDB$RELATIONS" Access By ID
            -> Bitmap
                -> Index "RDB$INDEX_0" Unique Scan
```

Scan for equality

An equality scan is used for the predicates `IS NULL`, `IS NOT DISTINCT FROM` and equality (`=`). If an equality scan involves all index segments, the Explain plan for such a scan will specify (`full match`), otherwise it will specify (`partial match`) and the number of segments involved.

In this case, the cardinality is calculated using the formula:

```
cardinality = table_cardinality * index_selectivity
```

Here `index_selectivity` is the selectivity of the set of index segments involved in the equality scan.

The cost of an equality index scan is calculated a little more complicatedly

```
cost = indexDepth + MAX(avgKeyLength * table_cardinality * index_selectivity / (page_size - BTR_SIZE), 1)
```

Where `avgKeyLength` is the average key length, `page_size` is the page size, `BTR_SIZE` is the index page header size, currently 39 bytes.

Currently, the average key length is not stored in the index statistics, but is calculated from the average compression ratio and the key length using the formula:

```
avgKeyLength = 2 + length * factor
```

Here `length` is the length of the index key, `factor` is the compression ratio.

The compression ratio is considered to be 0.5 for simple indexes, and 0.7 for composite ones.

Example 8. Index Range Scan (full match)

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

```
PLAN (RDB$RELATIONS INDEX (RDB$INDEX_1))
```

```
[cardinality=1.02, cost=4.020]
Select Expression
    [cardinality=1.02, cost=4.020]
    -> Filter
        [cardinality=1.02, cost=4.020]
        -> Table "RDB$RELATIONS" Access By ID
            -> Bitmap
                -> Index "RDB$INDEX_1" Range Scan (full match)
```

Example 9. Index Range Scan (full match) for a composite index

```
SELECT *
FROM EMPLOYEE
WHERE FIRST_NAME = ? AND LAST_NAME = ?
```

```
PLAN (EMPLOYEE INDEX (NAMEX))
```

```
[cardinality=1.0, cost=4.000]
Select Expression
    [cardinality=1.0, cost=4.000]
    -> Filter
        [cardinality=1.0, cost=4.000]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "NAMEX" Range Scan (full match)
```

Now let's look at an example where only the first of the two segments is used for the composite index.

Example 10. Index Range scan (partial match)

```
SELECT *
FROM EMPLOYEE
WHERE LAST_NAME = ?
```

```
PLAN (EMPLOYEE INDEX (NAMEX))
```

```
[cardinality=1.05, cost=4.050]
Select Expression
    [cardinality=1.05, cost=4.050]
    -> Filter
        [cardinality=1.05, cost=4.050]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "NAMEX" Range Scan (partial match: 1/2)
```

Range Scan with Bounds

If the upper and lower scan bounds do not match or only one of them is used, the Explain plan will display which scan bounds are used: "lower bound" — lower bound, "upper bound" — upper bound. For each bound, the number of index segments used is indicated. For example, (lower bound: 1/2) means that one segment out of two will be used for the lower bound.

The cost and cardinality of a range scan are calculated in the same way as for an equality scan, but instead of index selectivity, constants for predicate selectivity are used.

Table 1. Selectivity of index scan predicates

Predicates	Selectivity
<, <=, >, >=	0.05
BETWEEN	0.0025
STARTING WITH	0.01
IN	indexSelectivity * N

Example 11. Index Range Scan with lower bound

```
SELECT *
FROM EMPLOYEE
WHERE EMP_NO > 0
```

```
PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
```

```
[cardinality=6.09, cost=9.090]
Select Expression
    [cardinality=6.09, cost=9.090]
    -> Filter
        [cardinality=6.09, cost=9.090]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "RDB$PRIMARY7" Range Scan (lower bound: 1/1)
```

Example 12. Index Range Scan with upper bound

```
SELECT *
FROM EMPLOYEE
WHERE EMP_NO < 0
```

```
PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
```

```
[cardinality=6.09, cost=9.090]
Select Expression
    [cardinality=6.09, cost=9.090]
    -> Filter
        [cardinality=6.09, cost=9.090]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "RDB$PRIMARY7" Range Scan (upper bound: 1/1)
```

Example 13. Index Range Scan with lower and upper bound

```
SELECT *
FROM EMPLOYEE
WHERE EMP_NO BETWEEN ? AND ?
```

```
PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
```

```
[cardinality=4.295, cost=7.295]
Select Expression
    [cardinality=4.295, cost=7.295]
    -> Filter
        [cardinality=4.295, cost=7.295]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "RDB$PRIMARY7" Range Scan (lower bound: 1/1, upper bound: 1/1)
```

Now let's look at examples for composite indexes, where only part of the segments are used for boundaries.

Example 14. Range Scan of a composite index (lower bound uses the first of two segments)

```
SELECT *
FROM EMPLOYEE
WHERE LAST_NAME > ?
```

```
PLAN (EMPLOYEE INDEX (NAMEX))
```

```
[cardinality=3.09, cost=6.090]
Select Expression
    [cardinality=3.09, cost=6.090]
    -> Filter
        [cardinality=3.09, cost=6.090]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "NAMEX" Range Scan (lower bound: 1/2)
```

Example 15. Range Scan of a composite index. The lower bound uses both segments, and the upper bound uses only the first

```
SELECT *
FROM EMPLOYEE
WHERE LAST_NAME = ? AND FIRST_NAME > ?
```

```
PLAN (EMPLOYEE INDEX (NAMEX))
```

```
[cardinality=1.03, cost=4.030]
Select Expression
    [cardinality=1.03, cost=4.030]
    -> Filter
        [cardinality=1.03, cost=4.030]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "NAMEX" Range Scan (lower bound: 2/2, upper bound: 1/2)
```

Let's try the opposite:

```
SELECT *
FROM EMPLOYEE
WHERE LAST_NAME > ? AND FIRST_NAME = ?
```

```
PLAN (EMPLOYEE INDEX (NAMEX))
```

```
[cardinality=1.0, cost=6.097]
Select Expression
    [cardinality=1.0, cost=6.097]
    -> Filter
        [cardinality=3.097, cost=6.097]
        -> Table "EMPLOYEE" Access By ID
            -> Bitmap
                -> Index "NAMEX" Range Scan (lower bound: 1/2)
```

As you can see, only the lower boundary of the first segment is used, and the second segment is not involved at all.

Now let's try to involve only the second segment.

```
SELECT *
FROM EMPLOYEE
WHERE FIRST_NAME = ?
```

```
PLAN (EMPLOYEE NATURAL)
```

```
[cardinality=4.2, cost=42.000]
Select Expression
    [cardinality=4.2, cost=42.000]
    -> Filter
        [cardinality=42.0, cost=42.000]
        -> Table "EMPLOYEE" Full Scan
```

In this case, index scanning could not be used.

Some DBMS (notably Oracle) have a so-called Index Skip Scan, where the entire composite index key is not compared, but only the affected part of the key is compared, in which case the last example could use an index scan. This access method is currently not available in Firebird.

List scan

List scan is available since Firebird 5.0. It is applicable only for the `IN` predicate with a list of values. In this case, one common bitmap is formed for the entire list of values. The search can be performed by vertical scanning (from the root for each key) or horizontal scanning of the range between min/max keys. How exactly the scan will be performed is decided by the optimizer, depending on what is cheaper in terms of cost.

Before Firebird 5.0, such a predicate was transformed into a set of equality conditions combined via the `OR` predicate.

That is,

```
F IN (V1, V2, ... VN)
```

was transformed into

```
(F = V1) OR (F = V2) OR .... (F = VN)
```

In Firebird 5.0, this works differently. Lists of constants in `IN` are pre-evaluated as invariants and cached as a binary search tree, which speeds up the comparison if the condition needs to be checked for many records or if the list of values is long. If an index is applicable to the predicate, then a list scan of the index is used.

The cardinality of this access method is calculated by the formula:

```
cardinality = table_cardinality * MAX(index_selectivity * N, 1)
```

Where `N` is the number of elements in the `IN` list.

The cost of scanning an index over a list is calculated as follows:

```
cost = indexDepth + MAX(avgKeyLength * table_cardinality * index_selectivity * N, 1)
```

Example 16. List scan

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME IN ('RDB$RELATIONS', 'RDB$PROCEDURES', 'RDB$FUNCTIONS')
```

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

```
[cardinality=3.2, cost=6.200]
Select Expression
    [cardinality=3.2, cost=6.200]
    -> Filter
        [cardinality=3.2, cost=6.200]
        -> Table "RDB$RELATIONS" Access By ID
            -> Bitmap
                -> Index "RDB$INDEX_0" List Scan (full match)
```

2.2.5. Intersection (ANDs) and union (ORs) of bitmaps

As mentioned above, bitmaps can be bit ANDed and bit ORed, so the server can use multiple indexes for a single table. Bitmap intersections do not increase the resulting cardinality. For example, for the expression `F = 0 and ? = 0`, the second part is not indexed and is therefore checked after the index fetching, without affecting the final result. But bitmap merging increases the resulting cardinality, so only parts of a fully indexed predicate can be merged. That is, when moving the second part of the expression `F = 0 or ? = 0` to a higher level, it may turn out that all records had to be scanned. Therefore, the index for the F field will not be used in such an expression.

The selectivity of the intersection of two bitmaps is calculated using the formula:

```
(bestSel + (worstSel - bestSel) / (1 - bestSel) * bestSel) / 2
```

The selectivity of combining two bitmaps is calculated using the formula:

```
bestSel + worstSel
```

Since the cost of access via a record identifier is one, the total cost of access via a bitmap is equal to the total cost of the index lookup (for all indexes that form the bitmap) plus the resulting cardinality of the bitmap.

Let’s look at examples using multiple indexes.

Example 17. Intersection (ANDs) of bitmaps

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

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

```
[cardinality=1.0, cost=7.000]
Select Expression
    [cardinality=1.0, cost=7.000]
    -> Filter
        [cardinality=1.0, cost=7.000]
        -> Table "RDB$INDICES" Access By ID
            -> Bitmap And
                -> Bitmap
                    -> Index "RDB$INDEX_31" Range Scan (full match)
                -> Bitmap
                    -> Index "RDB$INDEX_41" Range Scan (full match)
```

Example 18. Union (ORs) of bitmaps

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

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

```
[cardinality=10.85, cost=16.850]
Select Expression
    [cardinality=10.85, cost=16.850]
    -> Filter
        [cardinality=10.85, cost=16.850]
        -> Table "RDB$INDICES" Access By ID
            -> Bitmap Or
                -> Bitmap
                    -> Index "RDB$INDEX_31" Range Scan (full match)
                -> Bitmap
                    -> Index "RDB$INDEX_41" Range Scan (full match)
```

2.2.6. Index navigation

Index navigation is nothing more than sequential scanning of index keys. And since for each record the server needs to fetch the record and check its visibility for our transaction, this operation is

quite expensive. That is why this access method is not used for regular selections (unlike Oracle, for example), but is used only in cases where it is justified.

Sorting using multiple indices is not supported.

There are two such cases today. The first is the calculation of aggregate functions `MIN/MAX`. Obviously, to calculate `MIN` it is enough to take the first key in the ASC index, and to calculate `MAX` — the first key in the DESC index. If after fetching a record it turns out that it is not visible to us, then we take the next key, and so on. The second is sorting or grouping records. This is indicated by the `ORDER BY` or `GROUP BY` clauses in the user's query. In this situation, we simply go through the index, selecting records as we scan. In both cases, fetching records is performed based on access via its identifier.

The unidirectionality of index navigation is described in Indexed data access.

There are several features that optimize this process. First, if there are restrictive predicates on the sort field, they create upper and/or lower scan bounds. That is, in the case of the query (`WHERE A > 0 ORDER BY A`), a partial index scan will be performed instead of a full one. This way, we reduce the cost of the scan itself. Second, if there are other restrictive predicates (not on the sort field), but optimized through the index, a complex mode of operation is enabled, where index scanning is combined with the use of a bitmap. Let's consider how this works. Let us assume that we have a query of the form (`WHERE A > 0 AND B > 0 ORDER BY A`). In this case, first a range scan is performed for the index by field `B` and a bitmap is created. Then the value `0` is set as the lower bound of the index scan by field `A`. Then we scan this index starting from the lower bound and for each record number extracted from the index we check its inclusion in the bitmap. And only in case of entry we perform fetch of records. This way we reduce expenses on fetch of data pages.

The main difference between index navigation and scanning (described in Range Scan) is the absence of a bitmap between index scanning and record access via its identifier. The reason is clear — sorting records by physical numbers is contraindicated in this case. From this we can conclude that the index is scanned as it is fetched from the client, and not "all at once" (as in the case of using a bitmap).

The cost of navigation is quite difficult to estimate. To calculate `MIN/MAX` in the vast majority of cases it will be equal to the height of the B+ tree (search for the first key) plus one (page fetch). The cost of such access is considered to be a negligible value, since in practice the described calculation of `MIN/MAX` will always be faster than alternative options. To estimate the cost of index navigation, one must take into account both the number and average width of index keys and the cardinality of the bitmap (if any), and also have an idea of the clustering factor of the index — the coefficient of correspondence between the location of keys and physical numbers of records. At the moment, the server does not calculate the cost of navigation, i.e. the rule-based strategy is used again. In the future, it is planned to use this approach only for `MIN/MAX` and `FIRST`, and rely on the cost in other cases.

The clustering factor of an index is not stored in the index statistics, but you can view it using the `gstat` utility.

Firebird 6.0 added a cost estimate for choosing between index navigation and external sorting.

In the Legacy plan, index navigation is indicated by the word “ORDER” followed by the name of the index (without parentheses, since there can only be one such index). The server also reports the indices that form the bitmap used for filtering. In this case, both words are present in the execution plan: first “ORDER”, then “INDEX”.

Example 19. Index navigation with full index scan

```
SELECT RDB$RELATION_NAME
FROM RDB$RELATIONS
ORDER BY RDB$RELATION_NAME
```

```
PLAN (RDB$RELATIONS ORDER RDB$INDEX_0)
```

```
[cardinality=265.0, cost=302.000]
Select Expression
    [cardinality=265.0, cost=302.000]
    -> Table "RDB$RELATIONS" Access By ID
        -> Index "RDB$INDEX_0" Full Scan
```

Example 20. Index navigation with a predicate that creates a lower bound on a scan

```
SELECT MIN(RDB$RELATION_NAME)
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME > ?
```

```
PLAN (RDB$RELATIONS ORDER RDB$INDEX_0)
```

```
[cardinality=1.0, cost=26.785]
Select Expression
    [cardinality=1.0, cost=26.785]
    -> Aggregate
        [cardinality=18.785, cost=26.785]
        -> Filter
            [cardinality=18.785, cost=26.785]
            -> Table "RDB$RELATIONS" Access By ID
                -> Index "RDB$INDEX_0" Range Scan (lower bound: 1/1)
```

Example 21. Index navigation with a bitmap

```
SELECT MIN(RDB$RELATION_NAME)
FROM RDB$RELATIONS
WHERE RDB$RELATION_ID > ?
```

```
PLAN (RDB$RELATIONS ORDER RDB$INDEX_0 INDEX (RDB$INDEX_1))
```

```
[cardinality=1.0, cost=159.000]
Select Expression
    [cardinality=1.0, cost=159.000]
    -> Aggregate
        [cardinality=125.0, cost=159.000]
        -> Filter
            [cardinality=125.0, cost=159.000]
            -> Table "RDB$RELATIONS" Access By ID
                -> Index "RDB$INDEX_0" Full Scan
                    -> Bitmap
                        -> Index "RDB$INDEX_1" Range Scan (lower bound: 1/1)
```

2.3. Access via RDB$DB_KEY

The RDB$DB_KEY access mechanism is primarily used by the server for internal purposes. However, it can also be used in user queries.

In terms of access methods, everything is simple here - a bitmap is created and a single record number is placed in it — the RDB$DB_KEY value. After that, the record is read using the access described above via the record identifier. Obviously, the cost of such access is equal to one. That is, we get something like pseudo-index access, which is reflected in the query execution plan.

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

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

```
[cardinality=1.0, cost=1.0]
Select Expression
    [cardinality=1.0, cost=1.0]
    -> Filter
        [cardinality=1.0, cost=1.0]
        -> Table "RDB$RELATIONS" Access By ID
            -> DBKEY
```

Reads via RDB$DB_KEY are reported as indexed in the statistics. The reason for this is clear from the description above — everything that is read via record ID access is considered indexed by the server. Not entirely correct, but a fairly harmless assumption.

2.4. External table scan

This type of data access is used only when working with external tables. In essence, it is an analogue of a full table scan. Records are read from the external file one by one, using the current pointer (offset). Page cache is not used. Indexing of external tables is not supported.

Due to the lack of alternative methods of accessing external data, the cost is not calculated and is not used. The cardinality of the selection from the external table is estimated as `File_size / Record_size`.

In the Legacy plan, reading from an external table is represented by the word “NATURAL”.

In the Explain plan, reading from an external table is shown as “Table Full Scan”.

Example 22. Reading an external table

```
SELECT *
FROM EXT_TABLE
```

```
PLAN (EXT_TABLE NATURAL)
```

```
[cardinality=1000.0, cost=???]
Select Expression
    [cardinality=1000.0, cost=???]
    -> Table "EXT_TABLE" Full Scan
```

2.5. Virtual table scan

This access method is used to read data from virtual tables (MON$ monitoring tables, SEC$ security virtual tables, and RDB$CONFIG virtual tables). The data of such tables is generated on the fly and cached in memory. For example, all monitoring tables are filled the first time any of them is accessed, and their data is saved until the end of the transaction. The RDB$CONFIG virtual table is also filled the first time it is accessed, and its data is saved until the end of the session.

Due to the lack of alternative methods to access virtual tables, the cost is not calculated or used. The cardinality of virtual tables is assumed to be 1000.

In the Legacy plan, reading from a virtual table is displayed as “NATURAL”.

In the Explain plan, reading from a virtual table is displayed as “Table Full Scan”.

Example 23. Reading the MON$ATTACHMENT virtual table

```
SELECT *
FROM MON$ATTACHMENTS
```

```
PLAN (MON$ATTACHMENTS NATURAL)
```

```
[cardinality=1000.0, cost=???]
Select Expression
    [cardinality=1000.0, cost=???]
    -> Table "MON$ATTACHMENTS" Full Scan
```

2.6. Local Table Full Scan

This data access method is available since Firebird 5.0. It is used to return inserted, modified or deleted records by DML statements containing the RETURNING clause and returning a cursor. Such statements include INSERT ⋯ SELECT ⋯ RETURNING, UPDATE⋯ RETURNING, DELETE⋯ RETURNING, MERGE⋯ RETURNING. The INSERT ⋯ VALUES .. RETURNING statement, which can insert and return only one record, does not apply to them.

Local temporary tables are created "on the fly" with the columns listed in the RETURNING clause and store their data and structure until the end of the row selection from the DML statement.

Future versions of Firebird plan to add the ability to declare and populate local temporary tables in the local variable declaration section of PSQL modules, as DECLARE LOCAL TABLE.

Due to the lack of alternative methods of accessing local tables, the cost is not calculated or used. The cardinality of local temporary tables is assumed to be 1000.

In Legacy and Explain plans, a read from a local temporary table is displayed separately from the main plan (as for subqueries), immediately after it. In a Legacy plan, a read from a local table is displayed as “NATURAL” and the table is named Local_Table. In an Explain plan, a read from a local temporary table is displayed as “Local Table Full Scan”.

Example 24. Using a local temporary table for a DML statement with the RETURNING *clause*

```
UPDATE COLOR
SET NAME = ?
WHERE NAME = ?
RETURNING CODE_COLOR, NAME, OLD.NAME AS OLD_NAME
```

```
PLAN (COLOR INDEX (UNQ_COLOR_NAME))
PLAN (Local_Table NATURAL)
```

```
[cardinality=1.0, cost=4.0]
Select Expression
    [cardinality=1.0, cost=4.0]
    -> Filter
        [cardinality=1.0, cost=4.0]
        -> Table "COLOR" Access By ID
            -> Bitmap
                -> Index "UNQ_COLOR_NAME" Unique Scan
[cardinality=1000.0, cost=???]
Select Expression
    [cardinality=1000.0, cost=???]
    -> Local Table Full Scan
```

2.7. Procedure Scan

This data access method is used when fetching from stored procedures that use the SUSPEND clause to return the result. In Firebird, a stored procedure is always a black box, about the internals and implementation details of which the server makes no assumptions. A procedure is always considered a non-deterministic data source, i.e. it can return different data for two subsequent calls made under equal conditions. In light of this, it is obvious that any kind of indexing of the procedure results is impossible. A procedural access method is also an analogue of a full table scan. For each fetch from a procedure, it is executed from the moment of the previous stop (the beginning of the procedure for the first fetch) until the next SUSPEND clause, after which its output parameters form a data string, which is returned from this access method.

Similar to the data access method for external tables, the cost of procedural access is also not calculated or taken into account, since there are no alternatives. For stored procedures, the cardinality is assumed to be 1000.

In the Legacy execution plan, the procedural is displayed as “NATURAL”.

In older versions of Firebird (before 3.0), the details (plans) of selections inside a procedure can be output. This allows you to estimate the query execution plan as a whole, taking into account the "internals" of all participating procedures, but formally this is incorrect.

Example 25. Procedure Scan

```
SELECT *
FROM PROC_TABLE
```

```
PLAN (PROC_TABLE NATURAL)
```

```
[cardinality=1000.0, cost=???]
Select Expression
    [cardinality=1000.0, cost=???]
    -> Procedure "PROC_TABLE" Scan
```

Chapter 3. Filters

This group of data access methods is a converter of the received data. The main difference between this group and others is that all filters have only one input. They do not change the width of the input stream, but only reduce its cardinality according to the rules determined by their function. The filter input can receive a data stream from both the primary and any other data source.

From an implementation perspective, filters have another common feature: they are not costed. That is, all filters are assumed to execute in zero time.

Below we will describe the existing implementations of filter methods in Firebird and the tasks they solve.

3.1. Predicate checking

This is probably the most commonly used case, and also the easiest to understand. This filter checks some logical condition for each record passed to it as input. If this condition is met, then the record is passed to the output unchanged, otherwise it is ignored. Depending on the input data and the logical condition, a fetch of one record from this filter can lead to one or more fetches from the input stream. Degenerate cases of validation filters are predefined conditions of the form (1 = 1) or (1 <> 1), which either simply pass the input data through themselves or discard it.

As the name suggests, these filters are used to evaluate predicates in `WHERE`, `HAVING`, `ON` and other clauses. In order to reduce the resulting cardinality of the selection, predicate checks are always placed as “lower” (“deeper”) as possible in the query execution tree. In the case of checking on a table field, the predicate check will be performed immediately after fetching a record from this table.

Each predicate has its own selectivity and thus leads to a decrease in the cardinality of the output stream. For non-indexed access, the predicates have the following selectivity:

Table 2. Selectivity of predicates

Predicate	Selectivity
`=, IS NULL, IS NOT DISTINCT FROM`	`0.1`
`<, <=, >, >=, <>, IS NOT NULL, IS DISTINCT FROM`	`0.5`
`BETWEEN`	`0.25`
`STARTING WITH, CONTAINING`	`0.5`
`IN (<list>)`	`0.1 * N`
`IN (<select>), EXISTS(<select>)`	`0.5`

The selectivity of predicates combined via the operator `AND` is multiplied, and via `OR` it is summed.

Thus, the cardinality of the output stream is calculated by the formula

```
cardinality = selectivity * inputCardinality
```

In the Legacy plan, predicate checking is not displayed.

In the Explain plan, predicate checking is displayed using the word “Filter”, but the predicate itself is not displayed.

Example 26. Checking the filter predicate

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$SYSTEM_FLAG = 0
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=35.0, cost=350.0]
Select Expression
    [cardinality=35.0, cost=350.0]
    -> Filter
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" Full Scan
```

Predicate checking also appears when using indexed access.

Example 27. Checking a filter predicate when using indexed access

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

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

```
[cardinality=1.0, cost=4.0]
Select Expression
    [cardinality=1.0, cost=4.0]
    -> Filter
        [cardinality=1.0, cost=4.0]
        -> Table "RDB$RELATIONS" Access By ID
            -> Bitmap
                -> Index "RDB$INDEX_0" Unique Scan
```

The last example may raise the question: why Filter is used if the predicate is implemented via `INDEX UNIQUE SCAN`. The fact is that Firebird implements fuzzy search in the index. That is, index scanning is only an optimization of predicate calculation and in some cases can return more records than required. That is why the server does not rely only on the result of index scanning and

checks the predicate again, after fetching the record. Note that in the vast majority of cases this does not incur noticeable overhead costs in terms of performance. In this case, checking the predicate does not change the cardinality estimate, since it has already been obtained during index access (filtered by index).

3.1.1. Checking invariant predicates

A predicate is invariant if its value does not depend on the fields of the filtered streams. The simplest example of invariant predicates are conditions like `1=0` and `1=1`. Invariant predicates can also contain parameter markers, for example `? = 1`.

Starting with Firebird 5.0, invariant and at the same time deterministic predicates are recognized by the optimizer and are evaluated once. Unlike regular filtering predicates, invariant predicates are evaluated as early as possible, their check is always placed as "higher" in the query execution tree as possible. If an invariant filtering predicate evaluates to `FALSE`, then retrieval of records from the underlying data sources is immediately stopped.

Unlike regular filtering predicates, filtering with an invariant predicate does not change the cardinality of the output stream.

In the Legacy plan, checking of invariant predicates is not displayed. In the Explain plan, the check for invariant predicates is displayed using the word "Filter (preliminary)", but the predicate itself is not displayed.

Example 28. Checking invariant predicates

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$SYSTEM_FLAG = 0
  AND 1 = ?
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=35.0, cost=350.0]
Select Expression
    [cardinality=35.0, cost=350.0]
    -> Filter (preliminary)
        [cardinality=35.0, cost=350.0]
        -> Filter
            [cardinality=350.0, cost=350.0]
            -> Table "RDB$RELATIONS" Full Scan
```

3.2. Sort

Also known as external sort, this filter is used by the optimizer when it is necessary to sort the input stream when index navigation is not possible (see Index navigation). Examples of its use: sorting or

grouping (in case there are no suitable indexes or indexes are not applicable to the input stream), building a B+ tree index, performing a DISTINCT operation, preparing data for a one-pass merge (see below), and others.

Since the input data is by definition unordered, it is obvious that the sort filter must fetch all the records from its input stream before it can emit any of them to the output. Thus, this filter can be considered a buffered data source.

External sorting is performed as follows. The set of input records is placed in an internal buffer, after which it is sorted by the quick sort algorithm and the block is moved to external memory. Then the next block is filled in the same way and the process continues until the end of the records in the input stream. After that, the filled blocks are read and a binary merge tree is built on them. When reading from the sort filter, the tree is parsed and the records are merged in one pass. External memory can be either virtual memory or disk space, depending on the server configuration file settings.

When performing an external sort, Firebird writes both key fields (those specified in the ORDER BY or GROUP BY clause) and non-key fields (all other fields referenced within the query) to sort blocks, which are either stored in memory or in temporary files. After the sort is complete, these fields are read back from the sort blocks.

Sorting has two modes of operation: “normal” and “truncation”. The first of them preserves records with duplicate sort keys, while the second causes duplicates to be removed. It is the “truncation” mode that implements the DISTINCT operation, for example.

The cost for external sorting is not calculated. In normal sorting mode, the output flow cardinality estimate does not change, in truncating mode, the cardinality is divided by 10.

In a Legacy execution plan, sorting is denoted by the word “SORT” followed by a description of the input stream in parentheses.

In the Explain of the execution plan, sorting is denoted by the word “Sort” for normal mode and “Unique Sort” for truncated mode. The length of the records to be sorted (record length) and the length of the sort key (key length) are then specified in parentheses. The records to be sorted always include the sort key.

You can estimate the memory consumption for sorting by multiplying the length of the records to be sorted (record length) by the cardinality of the stream to be sorted. By default, Firebird will try to sort in RAM, and once this memory is exhausted, the engine will start using temporary files. The amount of RAM available for sorting is set by the `TempCacheLimit` parameter. In the Classic architecture, this limit is set for each connection, and in the SuperServer and SuperClassic architectures, for all database connections.

Example 29. Using External Sorting in Normal Mode

```
SELECT RDB$RELATION_NAME, RDB$SYSTEM_FLAG
FROM RDB$RELATIONS
ORDER BY RDB$SYSTEM_FLAG
```

```
PLAN SORT (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Sort (record length: 284, key length: 8)
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" Full Scan
```

Example 30. Using External Sort in Truncate Mode

```
SELECT DISTINCT
    RDB$RELATION_NAME, RDB$SYSTEM_FLAG
FROM RDB$RELATIONS
```

```
PLAN SORT (RDB$RELATIONS NATURAL)
```

```
[cardinality=35.0, cost=350.0]
Select Expression
    [cardinality=35.0, cost=350.0]
    -> Unique Sort (record length: 284, key length: 264)
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" Full Scan
```

Example 31. Using External Sorting with Filtering

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

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

```
[cardinality=125.0, cost=159.000]
Select Expression
    [cardinality=125.0, cost=159.000]
    -> Sort (record length: 284, key length: 8)
        [cardinality=125.0, cost=159.000]
        -> Filter
            [cardinality=125.0, cost=159.000]
            -> Table "RDB$RELATIONS" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_0" Range Scan (lower bound: 1/1)
```

Possible redundant sorting options (for example, `DISTINCT` and `ORDER BY` on the same field) are eliminated by the optimizer and reduced to only the minimum necessary number of sortings.

3.2.1. Refetch

Starting with Firebird 4.0, there is a sorting optimization for wide data sets, which is displayed in the Explain plan as `Refetch`. A ``wide data set' is a data set in which the total length of the record fields is large.

When performing an external sort, Firebird writes both key fields (those specified in the ORDER BY or GROUP BY clause) and non-key fields (all other fields referenced within the query) to sort blocks, which are either stored in memory or in temporary files. After the sort is complete, these fields are read back from the sort blocks. This approach is generally considered faster because records are read from the temporary files in the order corresponding to the sorted records, rather than being selected randomly from the data page. However, if the non-key fields are large (e.g., using long VARCHARs), this increases the size of the sort blocks and thus results in more I/O to the temporary files.

An alternative approach (`Refetch`) is that only key fields and `DBKEY` of records of all participating tables are stored inside sort blocks, and non-key fields are extracted from data pages after sorting. This improves sort performance in case of long non-key fields. It is clear from the description of `Refetch` that it can be used only in case of normal sort mode, `Refetch` is not applicable for truncated mode, since `DBKEY` does not get into sort blocks in this mode.

For external sorting using `Refetch`, the cost is not calculated. The optimizer uses the `InlineSortThreshold` parameter to select the method by which the data will be sorted. The value

specified for `InlineSortThreshold` determines the maximum size of a sort record (in bytes) that can be stored inline, i.e. inside the sort block. Zero means that records are always re-fetched (Refetch). The optimal value for this parameter should be selected experimentally. The default value is 1000 bytes.

In the Legacy execution plan, Refetch and external sorting are displayed the same way, as `SORT`.

In the Explain execution plan, sorting using Refetch is displayed as a tree, where the root is “Refetch”, at the bottom level is “Sort” . Then in brackets are the length of the keys to be sorted + `DBKEY` of the tables used (record length) and the length of the sort key (key length).

Example 32. Optimizing External Sorting Using Refetch

```
SELECT *
FROM RDB$RELATIONS
ORDER BY RDB$SYSTEM_FLAG
```

```
PLAN SORT (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Refetch
        -> Sort (record length: 28, key length: 8)
            [cardinality=350.0, cost=350.0]
            -> Table "RDB$RELATIONS" Full Scan
```

From the Explain plan it is clear that first sorting by sort keys occurs, the keys themselves + DBKEY of the table get into the sort blocks, and then full records are extracted from the table in sorted order by DBKEY using the `Refetch` method.

Now let’s try to use `Refetch` for truncating sort mode.

Example 33. In truncated sort mode, Refetch is not used

```
SELECT DISTINCT *
FROM RDB$RELATIONS
```

```
PLAN SORT (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Unique Sort (record length: 1438, key length: 1412)
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" Full Scan
```

Here, although wide records need to be sorted, `Refetch` cannot be used, and so the usual external sorting is used.

3.3. Aggregation

This filter is used exclusively for calculating aggregate functions (`MIN`, `MAX`, `AVG`, `SUM`, `COUNT` ...), including cases of their use in groupings.

The implementation is quite trivial. All the listed functions require a single temporary register to store the current boundary (`MIN`/`MAX`) or accumulated (other functions) value. Then, for each record from the input stream, we check or sum this register. In the case of grouping, the algorithm is somewhat more complicated. To correctly calculate the function for each group, the boundaries between these groups must be clearly defined. The simplest solution is to guarantee the ordering of the input stream. In this case, we perform aggregation for the same grouping key, and after changing it, we output the result and continue with the next key. This approach is exactly what is used in Firebird — aggregation by groups strictly depends on the presence of sorting at the input. The input stream can be ordered using external sorting or index navigation (if possible). The Refetch method cannot be used to order the input stream for grouping.

There is an alternative way to calculate aggregates — hashing. In this case, sorting of the input stream is not required, but a hash function is applied to each grouping key and a register is stored for each key (more precisely, for each key collision) in a hash table. The advantage of this algorithm is that there is no need for additional sorting. The disadvantage is greater memory consumption for storing the hash table. This method usually wins over sorting when the selectivity of grouping keys is low (few unique values, therefore a small hash table). Aggregation by hashing is not available in Firebird.

Hash aggregation is planned to be implemented in future versions of Firebird.

When aggregate functions are used without grouping, the output cardinality is always one. When using grouping, the input cardinality is divided by 1000. The cost of aggregation is not calculated.

Grouping can also be used without aggregate functions, in which case it is analogous to `SELECT DISTINCT` for eliminating duplicate records, but unlike `DISTINCT` it can use both external sorting and index navigation.

The Legacy execution plan does not show aggregation.

The Explain plan shows aggregation using the word “Aggregate”.

Example 34. Calculating the MAX aggregate function

```
SELECT MAX(RDB$RELATION_ID)
FROM RDB$RELATIONS
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=1.0, cost=350.0]
Select Expression
    [cardinality=1.0, cost=350.0]
    -> Aggregate
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" Full Scan
```

Since there is an ASCENDING index on the `RDB$RELATION_ID` field, the `MIN` aggregate function can be calculated using index navigation (see Index navigation).

Example 35. Calculating the MIN aggregate function using index navigation

```
SELECT MIN(RDB$RELATION_ID)
FROM RDB$RELATIONS
```

```
PLAN (RDB$RELATIONS ORDER RDB$INDEX_1)
```

```
[cardinality=1.0, cost=4.0]
Select Expression
    [cardinality=1.0, cost=4.0]
    -> Aggregate
        [cardinality=1.0, cost=4.0]
        -> Table "RDB$RELATIONS" Access By ID
            -> Index "RDB$INDEX_1" Full Scan
```

Example 36. Calculate an aggregate function with grouping. External sorting is used to separate the groups.

```
SELECT RDB$SYSTEM_FLAG, COUNT(*)
FROM RDB$FIELDS
GROUP BY RDB$SYSTEM_FLAG
```

```
PLAN SORT (RDB$FIELDS NATURAL)
```

```
[cardinality=7.756, cost=7756.0]
Select Expression
    [cardinality=7.756, cost=7756.0]
    -> Aggregate
        [cardinality=7756.0, cost=7756.0]
        -> Sort (record length: 28, key length: 8)
            [cardinality=7756.0, cost=7756.0]
            -> Table "RDB$RELATIONS" Full Scan
```

Example 37. Calculate an aggregate function with grouping. Index navigation is used to separate groups.

```
SELECT RDB$RELATION_NAME, COUNT(*)
FROM RDB$RELATION_FIELDS
GROUP BY RDB$RELATION_NAME
```

```
PLAN (RDB$RELATION_FIELDS ORDER RDB$INDEX_4)
```

```
[cardinality=2.4, cost=4381.0]
Select Expression
    [cardinality=2.4, cost=4381.0]
    -> Aggregate
        [cardinality=2400.0, cost=4381.0]
        -> Table "RDB$RELATION_FIELDS" Access By ID
            -> Index "RDB$INDEX_4" Full Scan
```

Example 38. Calculating an aggregate function with grouping and filtering

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

```
PLAN (RDB$RELATION_FIELDS ORDER RDB$INDEX_4)
```

```
[cardinality=0.13, cost=237.3]
Select Expression
    [cardinality=0.13, cost=237.3]
    -> Aggregate
        [cardinality=130.1, cost=237.3]
        -> Filter
            [cardinality=130.1, cost=237.3]
            -> Table "RDB$RELATION_FIELDS" Access By ID
                -> Index "RDB$INDEX_4" Range Scan (lower bound: 1/1)
```

When calculating the SUM/AVG/COUNT functions in DISTINCT mode, each group of values is sorted in the duplicate elimination mode before accumulation. In this case, the word SORT is not displayed in the plan.

Example 39. Calculating an aggregate function with grouping and filtering

```
SELECT RDB$RELATION_NAME, COUNT(DISTINCT RDB$NULL_FLAG)
FROM RDB$RELATION_FIELDS
GROUP BY RDB$RELATION_NAME
```

```
PLAN (RDB$RELATION_FIELDS ORDER RDB$INDEX_4)
```

```
[cardinality=2.4, cost=4381.0]
Select Expression
    [cardinality=2.4, cost=4381.0]
    -> Aggregate
        [cardinality=2400.0, cost=4381.0]
        -> Table "RDB$RELATION_FIELDS" Access By ID
            -> Index "RDB$INDEX_4" Full Scan
```

3.3.1. Filtering in the HAVING clause

Predicates in the HAVING clause may contain aggregate functions or fields and grouping expressions. If the predicate is applied to an aggregate function, filtering occurs after grouping and aggregate

calculation. If the predicate is applied to fields and grouping expressions, the predicate is pushed inward so that it is evaluated before grouping and aggregation.

Example 40. Filtering by aggregate function in HAVING clause

```
SELECT RDB$RELATION_NAME, COUNT(*)
FROM RDB$RELATION_FIELDS
GROUP BY RDB$RELATION_NAME
HAVING COUNT(*) > ?
```

```
PLAN (RDB$RELATION_FIELDS ORDER RDB$INDEX_4)
```

```
[cardinality=1.2, cost=4381.0]
Select Expression
    [cardinality=1.2, cost=4381.0]
    -> Filter
        [cardinality=2.4, cost=4381.0]
        -> Aggregate
            [cardinality=2400.0, cost=4381.0]
            -> Table "RDB$RELATION_FIELDS" Access By ID
                -> Index "RDB$INDEX_4" Full Scan
```

Example 41. Filter by grouping column

```
SELECT RDB$RELATION_NAME, COUNT(*)
FROM RDB$RELATION_FIELDS
GROUP BY RDB$RELATION_NAME
HAVING RDB$RELATION_NAME > ?
```

```
PLAN (RDB$RELATION_FIELDS ORDER RDB$INDEX_4)
```

```
[cardinality=1.0, cost=237.3]
Select Expression
    [cardinality=1.0, cost=237.3]
    -> Filter
        [cardinality=0.13, cost=237.3]
        -> Aggregate
            [cardinality=130.1, cost=237.3]
            -> Filter
                [cardinality=130.1, cost=237.3]
                -> Table "RDB$RELATION_FIELDS" Access By ID
                    -> Index "RDB$INDEX_4" Range Scan (lower bound: 1/1)
```

In the last example, the filtering predicate was "pushed down", but after the aggregates were

calculated, the filtering was applied again. Note that the re-filtering performed cardinality normalization (in theory, aggregate functions with or without grouping can never return less than one record).

3.4. Counters

This type of filter is also very simple. Its purpose is to return only a part of the input stream records, based on some value N of the internal counter. There are two types of this filter, used to implement the FIRST/SKIP/ROWS/FETCH FIRST/OFFSET clauses of the query. The first type (FIRST counter) returns only the first N records of its input stream, after which it returns the EOF flag. The second type (SKIP counter) ignores the first N records of its input stream and starts returning them to the output, starting with record N+1. Obviously, if the query has a selection constraint of the type (`FIRST 100 SKIP 100`), the SKIP counter should be applied first, and only after it — the FIRST counter. The optimizer guarantees the correct application of counters when executing the query.

The SKIP counter does not change the output flow cardinality. The output flow cardinality after the FIRST counter is equal to the value specified for this counter, if the value is not a literal, then the cardinality is assumed to be 1000.

The Legacy execution plan does not display counters.

In the Explain plan, the FIRST counter is displayed as “First N Records” and the SKIP counter is displayed as “Skip N Records”.

Example 42. Using the FIRST counter

```
SELECT *
FROM RDB$RELATIONS
FETCH FIRST 10 ROWS ONLY
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=10.0, cost=350.5]
Select Expression
    [cardinality=10.0, cost=350.5]
    -> First N Records
        [cardinality=350.5, cost=350.5]
        -> Table "RDB$RELATIONS" Full Scan
```

Example 43. Using FIRST(?)

```
SELECT FIRST(?) *
FROM RDB$RELATIONS
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=1000.0, cost=350.5]
Select Expression
    [cardinality=1000.0, cost=350.5]
    -> First N Records
        [cardinality=350.5, cost=350.5]
        -> Table "RDB$RELATIONS" Full Scan
```

Example 44. Using SKIP counter

```
SELECT *
FROM RDB$RELATIONS
OFFSET 10 ROWS
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.5, cost=350.5]
Select Expression
    [cardinality=350.5, cost=350.5]
    -> Skip N Records
        [cardinality=350.5, cost=350.5]
        -> Table "RDB$RELATIONS" Full Scan
```

Example 45. Simultaneous use of FIRST and SKIP counters

```
SELECT FIRST(10) SKIP(20) *
FROM RDB$RELATIONS
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=10.0, cost=350.5]
Select Expression
    [cardinality=10.0, cost=350.5]
    -> First N Records
        [cardinality=350.5, cost=350.5]
        -> Skip N Records
            [cardinality=350.5, cost=350.5]
            -> Table "RDB$RELATIONS" Full Scan
```

3.5. Singularity Check

This filter is used to ensure that only one record is returned from the data source. It is used when singular subqueries are used in the query text.

To perform its function, the singularity check filter performs two reads from the input stream. If the second read returns EOF, then we output the first record. Otherwise, we raise the error `isc_sing_select_err` (“multiple rows in singleton select”).

The cardinality of the output stream after the singularity check is 1.

The singularity check is not displayed in the Legacy execution plan.

The singularity check is displayed as “Singularity Check” in the Explain plan.

Example 46. Singularity checking of a correlated subquery

```
SELECT
  (SELECT RDB$RELATION_NAME FROM RDB$RELATIONS R
   WHERE R.RDB$RELATION_ID = D.RDB$RELATION_ID - 1) AS LAST_RELATION,
  D.RDB$SQL_SECURITY
FROM RDB$DATABASE D
```

```
PLAN (R INDEX (RDB$INDEX_1))
PLAN (D NATURAL)
```

```
[cardinality=1.0, cost=4.62]
Sub-query
    [cardinality=1.0, cost=4.62]
    -> Singularity Check
        [cardinality=1.62, cost=4.62]
        -> Filter
            [cardinality=1.62, cost=4.62]
            -> Table "RDB$RELATIONS" as "R" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_1" Range Scan (full match)
[cardinality=1.0, cost=5.62]
Select Expression
    [cardinality=1.0, cost=1.0]
    -> Table "RDB$DATABASE" as "D" Full Scan
```

In Legacy form, the first plan refers to the subquery, and the second to the main query.

Example 47. Singularity Check for uncorrelated subquery

```
SELECT *
FROM RDB$RELATIONS
WHERE RDB$RELATION_ID = ( SELECT RDB$RELATION_ID - 1 FROM RDB$DATABASE )
```

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

```
[cardinality=1.0, cost=1.0]
Sub-query (invariant)
    [cardinality=1.0, cost=1.0]
    -> Singularity Check
        [cardinality=1.0, cost=1.0]
        -> Table "RDB$DATABASE" Full Scan
[cardinality=1.62, cost=5.62]
Select Expression
    [cardinality=1.62, cost=4.62]
    -> Filter
        [cardinality=1.62, cost=4.62]
        -> Table "RDB$RELATIONS" Access By ID
            -> Bitmap
                -> Index "RDB$INDEX_1" Range Scan (full match)
```

3.6. Write Lock

This filter implements pessimistic record locking. It is applied only if the query text contains the clause `WITH LOCK`. Each record read from the filter input stream is returned to the output already locked by the current transaction. Locking is performed by creating a new version of the record, marked with the identifier of the current transaction. If the current transaction has already modified this record, then locking is not performed.

In current versions, the lock filter works only for primary access methods. It does not change the cardinality of the output data stream. The lock is not displayed in the Legacy execution plan.

Example 48. Блокировка записи

```
SELECT *
FROM COLOR
WITH LOCK
```

```
PLAN (COLOR NATURAL)
```

```
[cardinality=233.0, cost=233.0]
Select Expression
    [cardinality=233.0, cost=233.0]
    -> Write Lock
        [cardinality=233.0, cost=233.0]
        -> Table "COLOR" Full Scan
```

3.7. Conditional Stream

Conditional branching of streams is available since Firebird 3.0. This access method is used in cases where, depending on the input parameter, the table can be read either by a full scan or by using index access. Typically, this is possible for a filter condition like `INDEXED_FIELD = ? OR 1=?`.

The output stream cardinality is calculated as the average between the cardinalities of the data extraction options. The cost is also calculated as the average cost between the first and second options. It is assumed that the probability of executing different options is the same.

```
cardinality = (cardinality_option_1 + cardinality_option_2) / 2

cost = (cost_option_1 + cost_option_2) / 2
```

In the Legacy plan, the conditional branching is not displayed, but both options for retrieving data from the table are displayed, separated by commas.

In the Explain plan, conditional branching is displayed as a tree whose root is denoted by the word `Condition`, and the leaves of the tree are the options for retrieving data from the table.

Example 49. Conditional branching of streams

```
SELECT *
FROM RDB$RELATIONS R
WHERE (R.RDB$RELATION_NAME = ? OR 1=?)
```

```
PLAN (R NATURAL, R INDEX (RDB$INDEX_0))
```

```
[cardinality=175.65, cost=177.15]
Select Expression
    [cardinality=175.65, cost=177.15]
    -> Filter
        [cardinality=175.65, cost=177.15]
        -> Condition
            [cardinality=350.0, cost=350.0]
            -> Table "RDB$RELATIONS" as "R" Full Scan
            [cardinality=1.3, cost=4.3]
            -> Table "RDB$RELATIONS" as "R" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_0" Unique Scan
```

3.8. Record Buffer

This access method is used as an auxiliary one when implementing other higher-level access methods, such as “Sliding Window” and “Hash Join”.

Record buffering is intended to store the results of underlying access methods in RAM, once this memory is exhausted, the engine will start using temporary files. The amount of internal memory available for record buffering is set by the `TempCacheLimit` parameter. In the Classic architecture, this limit is set for each connection, and in the Classic and SuperClassic architectures for all database connections.

The memory consumption for buffering can be estimated by multiplying the record length by the cardinality of the input stream.

In the Legacy plan, record buffering is not displayed.

In the Explain plan, record buffering is shown as “Record Buffer”, followed by the record length in parentheses as (`record length: <length>`).

Examples that use record buffering will be shown in the description of the “Sliding Window” and “Hash Join” access methods.

3.9. Sliding Window

This group of access methods is used when calculating so-called window or analytic functions.

A window function performs calculations on a set of rows that are somehow related to the current row. Its action can be compared to the calculation performed by an aggregate function. However, with window functions, the rows are not grouped into a single output row, which is the case with regular, non-window, aggregate functions. Instead, these rows remain separate entities. Internally, a window function, like an aggregate function, can access more than just the current row of the query result. The set of rows over which the window function performs calculations is called a **window**.

A set of rows can be partitioned. By default, all rows in a window are included in a single section; this can be changed by specifying in `PARTITION BY` how the rows are partitioned. For each row, the window function only considers the rows that fall into the same section as the current row.

Within each partition, you can specify the order in which the window function processes the rows. This can be done in the window expression using `ORDER BY`.

Additionally, a window expression can specify a window frame, which determines how many rows in a section will be processed. By default, the window frame depends on the type of window function. Most often, if the order of row processing is specified (`ORDER BY` in the window clause), it is from the beginning of the section to the current row or value.

The syntax of window functions and window expressions is described in the chapter “Window (analytic) functions” of the "Firebird SQL Language Reference".

It follows from the above that the access methods used to calculate window functions do not change the cardinality of the selection. The cost of access methods for calculating window functions is not calculated, since there are no alternatives.

Evaluation of window functions begins with a result buffer, which is executed after all other parts of the `SELECT` query, but before sorting (`ORDER BY`), limiting the result using first/skip counters, and evaluating expressions for columns in the `SELECT` clause. This buffer is called the window buffer. In the Explain plan, it is referred to as the `Window Buffer` (see also Record Buffer).

Next, the partition boundaries are determined in the window buffer, which is designated as `Window Partition` in the Explain plan. This is done even if the `PARTITION BY` clause is missing (there will be one partition).

If the window expression describes partitioning with `PARTITION BY`, the original (one partition) will be partitioned again. external sort by partition keys is used to partition. External sort is also used to determine the order of row processing (`ORDER BY` inside the window). Firebird combines partition keys and keys for determining the order of rows into one external sort. If the set of rows for external sorting is too wide, the Refetch method can be used for optimization.

After partitioning and sorting the rows, the section above the resulting set is buffered again using the Record Buffer access method, over which the `Window Partition` partitioning method will be used.

There will be as many `Window Partition` + `Record Buffer` and possibly external sortings as there are different windows used in the query.

Once all partitions are defined and the rows within the partitions are ordered, the window

functions themselves are evaluated, which is displayed in the Explain plan as `Window`. During the evaluation of window functions within a section, the set of rows for the function evaluation can either grow from the beginning of the section (`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`), or decrease when moving to the end of the section (`ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`), or have a fixed size (move along the section relative to the current row) (`ROWS BETWEEN BETWEEN 2 PRECEDING AND 2 FOLLOWING`), or remain unchanged if the window frame is defined as `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`). That is why this access method is called “Sliding window”.

In the Legacy plan, the calculation of window functions is not displayed in any way, but if external sorting (SORT) is used to separate window sections or order rows within a section, it will be displayed.

Example 50. Calculating the simplest window function

```
SELECT
  RDB$RELATION_NAME,
  COUNT(*) OVER() AS CNT
FROM RDB$RELATIONS
```

```
PLAN (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Window
        [cardinality=350.0, cost=350.0]
        -> Window Partition
            [cardinality=350.0, cost=350.0]
            -> Window Buffer
                [cardinality=350.0, cost=350.0]
                -> Record Buffer (record length: 273)
                    [cardinality=350.0, cost=350.0]
                    -> Table "RDB$RELATIONS" Full Scan
```

There is only one partition here, into which all the query records are sent.

Example 51. Calculating the partitioned window function

```
SELECT
  RDB$RELATION_NAME,
  COUNT(*) OVER(PARTITION BY RDB$SYSTEM_FLAG) AS CNT
FROM RDB$RELATIONS
```

```
PLAN SORT (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Window
        [cardinality=350.0, cost=350.0]
        -> Window Partition
            [cardinality=350.0, cost=350.0]
            -> Record Buffer (record length: 546)
                [cardinality=350.0, cost=350.0]
                -> Sort (record length: 556, key length: 8)
                    [cardinality=350.0, cost=350.0]
                    -> Window Partition
                        [cardinality=350.0, cost=350.0]
                        -> Window Buffer
                            [cardinality=350.0, cost=350.0]
                            -> Record Buffer (record length: 281)
                                [cardinality=350.0, cost=350.0]
                                -> Table "RDB$RELATIONS" Full Scan
```

We add the order of processing rows in the partition, which will also implicitly set the window frame (`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`).

Example 52. Calculating a partitioned window function with row ordering

```
SELECT
  RDB$RELATION_NAME,
  COUNT(*) OVER(PARTITION BY RDB$SYSTEM_FLAG ORDER BY RDB$RELATION_ID) AS CNT
FROM RDB$RELATIONS
```

```
PLAN SORT (RDB$RELATIONS NATURAL)
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Window
        [cardinality=350.0, cost=350.0]
        -> Window Partition
            [cardinality=350.0, cost=350.0]
            -> Record Buffer (record length: 546)
                [cardinality=350.0, cost=350.0]
                -> Sort (record length: 564, key length: 16)
                    [cardinality=350.0, cost=350.0]
                    -> Window Partition
                        [cardinality=350.0, cost=350.0]
                        -> Window Buffer
                            [cardinality=350.0, cost=350.0]
                            -> Record Buffer (record length: 281)
                                [cardinality=350.0, cost=350.0]
                                -> Table "RDB$RELATIONS" Full Scan
```

Now let’s see how the plan changes if several window functions with different windows are used.

Example 53. Calculating multiple window functions with different windows

```
SELECT
  RDB$RELATION_NAME,
  COUNT(*) OVER W1 AS CNT,
  LAG(RDB$RELATION_NAME) OVER W2 AS PREV
FROM RDB$RELATIONS
WINDOW
  W1 AS (PARTITION BY RDB$SYSTEM_FLAG ORDER BY RDB$RELATION_ID),
  W2 AS (ORDER BY RDB$RELATION_ID)
```

```
PLAN SORT (SORT (RDB$RELATIONS NATURAL))
```

```
[cardinality=350.0, cost=350.0]
Select Expression
    [cardinality=350.0, cost=350.0]
    -> Window
        [cardinality=350.0, cost=350.0]
        -> Window Partition
            [cardinality=350.0, cost=350.0]
            -> Record Buffer (record length: 579)
                [cardinality=350.0, cost=350.0]
                -> Sort (record length: 588, key length: 8)
                    [cardinality=350.0, cost=350.0]
                    -> Window Partition
                        [cardinality=350.0, cost=350.0]
                        -> Record Buffer (record length: 546)
                            [cardinality=350.0, cost=350.0]
                            -> Sort (record length: 564, key length: 16)
                                [cardinality=350.0, cost=350.0]
                                -> Window Partition
                                    [cardinality=350.0, cost=350.0]
                                    -> Window Buffer
                                        [cardinality=350.0, cost=350.0]
                                        -> Record Buffer (record length: 281)
                                            [cardinality=350.0, cost=350.0]
                                            -> Table "RDB$RELATIONS" Full Scan
```

Chapter 4. Merging methods

This category of access methods is very similar to filters. Merge methods also transform input data using a specific algorithm. The only formal difference is that this group of methods always operates with several input streams. In addition, the usual result of their work is either expanding the selection by fields or increasing its cardinality.

There are two classes of merge methods — join and union — that perform different functions. In addition, this category also includes the execution of recursive queries, which is a special case of unions. Below we will get acquainted with their description and possible implementation algorithms.

4.1. Join

As you can guess from the name, this group of methods implements SQL joins. The SQL standard defines two types of joins: inner and outer. In addition, outer joins are divided into one-way (left or right) and full. Any type of join has two input streams - left and right. For inner and full outer joins, these streams are semantically equivalent. In the case of a one-way outer join, one of the streams is the leading (mandatory), and the second is the slave (optional). Often, the leading stream is also called outer, and the slave is inner. For a left outer join, the leading (outer) stream is the left one, and the slave (inner) stream is the right one. For a right outer join, it is vice versa.

Let me note right away that formally, a left outer join is equivalent to an inverted right outer join. Inversion in this context means replacing an outer flow with an inner one or vice versa. So it is enough to implement only one type of one-way outer join (usually left). This is what is done in Firebird. However, the decision to convert a right join to a left (basic) one and its subsequent optimization was made by the optimizer, which often led to incorrect optimization of right joins. Starting with version 1.5, the server converts right joins to left ones at the BLR parsing level, which eliminates possible conflicts in the optimizer.

Each join, in addition to the input streams, has another attribute — the join condition. It is this condition that determines the result, i.e. how exactly the input stream data will be matched. In the absence of this condition, we get a degenerate case — the Cartesian product (cross join) of the input streams.

Let's return to one-way outer joins. Their streams are called leading and trailing for a reason. In this case, the outer (leading) stream must always be read before the inner (trailing) one, otherwise it will be impossible to perform the required standard substitution of NULL values in the event of no correspondence between the inner stream and the outer one. From this we can conclude that the optimizer has no ability to choose the order of execution of a one-way outer join and it will always be determined by the query text. While for inner and full outer joins, the input streams are independent and can be read in any order, therefore the algorithm for executing such joins is determined exclusively by the optimizer and does not depend on the query text.

In addition to the joins provided in the SQL standard, many DBMSs also implement additional join methods: semi-join and anti-join. These join types are not directly provided in SQL, but can be used by the optimizer after transforming the SQL query execution tree. For example, they can be used to implement the filtering predicates `EXISTS` and `NOT EXISTS`. These join types are asymmetrical in

nature, i.e. the joined streams are not equivalent for them — they distinguish between the leading (mandatory) and the slave (the one that is checked in the filtering predicate).

4.1.1. Nested Loop Join

This method is the most common in Firebird. In other DBMSs, this algorithm is also called nested loops join.

Its essence is simple. One of the input streams (outer) is opened and one record is read from it. After that, the second stream (inner) is opened and one record is also read from it. Then the join condition is checked. If the condition is met, these two records are merged into one and output. Then the second record is read from the inner stream and the process is repeated until EOF is output from it. If the inner stream does not contain any records corresponding to the outer stream, then two scenarios are possible. In the case of an inner join, the record is not returned to the output. In the case of an outer join, we join the record from the outer stream with the required number of NULL values and output it. Next, regardless of the type of join, we read the second record from the outer stream and begin the iteration process over the inner stream again. It is obvious that this algorithm works on a pipeline principle and the join of streams is performed directly in the process of the client fetch.

The key feature of this join method is the "nested" selection from the inner stream. It is obvious that reading the entire inner stream for each record of the outer stream is very expensive. Therefore, nested loop join works efficiently only if there is an index applicable to the join condition. In this case, at each iteration, a subset of records that satisfy the current record of the outer stream will be selected from the inner stream. It is worth noting that not every index is suitable for efficient execution of this algorithm, but only the one that corresponds to the join condition. For example, with a join of the form (`ON SLAVE.F = 0`), even if an index is used on the `F` field of the inner table, the same records will still be read from it at each iteration, which is a waste of resources. In this situation, a MERGE/HASH join would be more efficient (see below).

We can introduce the definition of "stream dependency" as the presence of fields of both streams in the join condition and conclude that joining with nested loops is effective only when streams depend on each other.

If we recall the description of "Predicate checking", where the principle of maximally “deep” placement of predicative filters by the optimizer is described, then one point becomes clear: individual table filters will push “down” the join methods. Accordingly, in the case of a predicate of the type (`WHERE MASTER.F = 0`) and the absence of records in the `MASTER` table with the `F` field equal to zero, there will be no calls to the inner join stream at all, since in this case there is no iteration over the outer stream (there are no records in it).

Since the logical `AND` and `OR` connections are optimized via bitmaps, the nested loop join can be used for all kinds of join conditions, and is usually quite efficient.

For outer joins, the nested loop join is always selected because there are currently no alternatives. For inner joins, the optimizer selects the nested loop join if its cost is cheaper than the alternatives (Hash Join). If there are no suitable indices for the link fields of inner joins, then alternative join algorithms are almost always selected if they are possible. In addition, for inner joins, the optimizer selects the most efficient order of streams. The main selection criteria are the cardinalities of both

streams and the selectivity of the link condition. The following formulas are used to estimate the cost of a certain join order:

```
cost = cardinality(outer) + cardinality(outer) * (indexScanCost + cardinality(inner) *
selectivity(link))

cardinality = cardinality(outer) * (cardinality(inner) * selectivity(link))
```

The last part of the formula defines the cost of selection from the inner stream at each iteration. Multiplying it by the number of iterations yields the total cost of selection from the inner stream. The total cost is obtained by adding the cost of selection from the outer stream. Of all possible permutations, the one with the lowest cost is selected. In the process of enumerating the options, the obviously worst ones are discarded (based on the already available cost information). Permutations involve not only streams joined by nested loops, but also other join algorithms (Hash Join), the cost of which is calculated according to other rules.

In the Legacy execution plan, nested loops are joined by the word "JOIN", followed by parentheses and commas that describe the input streams.

In the Explain execution plan, nested loops are displayed as a tree, the root of which is labeled “Nested Loop Join” followed by the type of join in parentheses. The streams being joined are described at the level below.

Nested Loop Join (inner)

Example 54. Nested Loop Join (inner)

```
SELECT *
FROM RDB$RELATIONS R
JOIN RDB$RELATION_FIELDS RF ON R.RDB$RELATION_NAME = RF.RDB$RELATION_NAME
```

```
PLAN JOIN (R NATURAL, RF INDEX (RDB$INDEX_4))
```

```
[cardinality=3212.6, cost=4620.0]
Select Expression
    [cardinality=3212.6, cost=4620.0]
    -> Nested Loop Join (inner)
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" as "R" Full Scan
        [cardinality=9.2, cost=12.2]
        -> Filter
            [cardinality=9.2, cost=12.2]
            -> Table "RDB$RELATION_FIELDS" as "RF" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_4" Range Scan (full match)
```

Note one point. Since all streams are equivalent for an inner join, the optimizer converts the binary tree into a "flat" form in the case of more than two streams. It turns out that de facto an inner join operates with more than two input streams. This does not affect the algorithm in any way, but it explains the difference in the syntax of plans for inner and outer joins.

Example 55. Inner nested loop join of multiple streams

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

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

```
[cardinality=3212.6, cost=20152.4]
Select Expression
    [cardinality=3212.6, cost=20152.4]
    -> Nested Loop Join (inner)
        [cardinality=2428.0, cost=2428.0]
        -> Table "RDB$RELATION_FIELDS" as "RF" Full Scan
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "RDB$FIELDS" as "F" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_2" Unique Scan
        [cardinality=1.3, cost=4.3]
        -> Filter
            [cardinality=1.3, cost=4.3]
            -> Table "RDB$RELATIONS" as "R" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_0" Unique Scan
```

Cross-join is also an inner join and is always performed with nested loops.

Example 56. Nested Loop Join for CROSS JOIN

```
SELECT *
FROM RDB$PAGES
CROSS JOIN RDB$PAGES
```

```
PLAN JOIN (RDB$PAGES NATURAL, RDB$PAGES NATURAL)
```

```
[cardinality=261376.5625, cost=261376.5625]
Select Expression
    [cardinality=261376.5625, cost=261376.5625]
    -> Nested Loop Join (inner)
        [cardinality=511.25, cost=511.25]
        -> Table "RDB$PAGES" Full Scan
        [cardinality=511.25, cost=511.25]
        -> Table "RDB$PAGES" Full Scan
```

Nested Loop Join (outer)

Example 57. Nested Loop Join for OUTER JOIN

```
SELECT *
FROM RDB$RELATIONS R
LEFT JOIN RDB$RELATION_FIELDS RF ON R.RDB$RELATION_NAME = RF.RDB$RELATION_NAME
```

```
PLAN JOIN (R NATURAL, RF INDEX (RDB$INDEX_4))
```

```
[cardinality=3200.6, cost=4620.0]
Select Expression
    [cardinality=3200.6, cost=4620.0]
    -> Nested Loop Join (outer)
        [cardinality=350.0, cost=350.0]
        -> Table "RDB$RELATIONS" as "R" Full Scan
        [cardinality=9.2, cost=12.2]
        -> Filter
            [cardinality=9.2, cost=12.2]
            -> Table "RDB$RELATION_FIELDS" as "RF" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_4" Range Scan (full match)
```

Example 58. Nested Loop Join for OUTER JOIN of multiple streams

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

```
PLAN JOIN (JOIN (R NATURAL, RF INDEX (RDB$INDEX_4)), F INDEX (RDB$INDEX_2))
```

```
[cardinality=3200.6, cost=16003.0]
Select Expression
    [cardinality=3200.6, cost=16003.0]
    -> Nested Loop Join (outer)
        [cardinality=3200.6, cost=4620.0]
        -> Nested Loop Join (outer)
            [cardinality=350.0, cost=350.0]
            -> Table "RDB$RELATIONS" as "R" Full Scan
            [cardinality=9.2, cost=12.2]
            -> Filter
                [cardinality=9.2, cost=12.2]
                -> Table "RDB$RELATION_FIELDS" as "RF" Access By ID
                    -> Bitmap
                        -> Index "RDB$INDEX_4" Range Scan (full match)
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "RDB$FIELDS" as "F" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_2" Unique Scan
```

Note that unlike the inner join, the Explain plan here contains nested “Nested Loop Join”.

Nested Loop Join (semi)

This type of join is currently not used by the Firebird optimizer.

Its essence is as follows. One of the input streams (outer) is opened and one record is read from it. After that, the second stream (inner) is opened and one record is also read from it. Then the join condition is checked. If the condition is match, the record from the outer stream is output, and the inner stream is immediately closed. If the join condition is not match, the second record is read from the inner stream and the process is repeated until EOF is issued from it. Then the outer stream reads the next record and everything is repeated until EOF is issued from it.

This data access method can be used to execute subqueries in `EXIST` and `IN (<select>)` predicates, as well as other predicates that evaluate to `EXIST` (`ANY`, `SOME`).

The cardinality of the output stream is very difficult to estimate, but what can be said for sure is that it will never exceed the cardinality of the outer stream. It is assumed that half of the records of

the outer stream match the join condition.

```
cardinality = cardinality(outer) * 0.5

cost = cardinality(outer) + cardinality(outer) * cost(inner first row)
```

Here `cost(inner first row)` is the cost of selecting the first record that matches the relation condition. Since the `EXISTS` predicate may contain a complex subquery, rather than reading from a single table, it is not so easy to calculate. It is also quite obvious that for this algorithm to work effectively, the optimizer must switch to the `FIRST ROWS` optimization strategy for executing the inner subquery.

Nested Loop Join (anti)

This type of join is currently not used by the Firebird optimizer.

In essence, this type of join is the opposite of “Nested Loop Join (semi)”. One of the input streams (outer) is opened and one record is read from it. After that, the second stream (inner) is opened and one record is also read from it. Then the join condition is checked. If the condition is not match, the record from the outer stream is output, and the inner stream is immediately closed. If the join condition is match, the second record is read from the inner stream and the process is repeated until EOF is issued from it. Then the outer stream reads the next record and everything is repeated until EOF is issued from it.

This data access method can be used to execute subqueries in the `NOT EXIST` predicate, as well as other predicates that are converted to `NOT EXIST (ALL)`. In addition, some one-way outer joins can also be converted to it under certain conditions.

The cardinality of the output stream is very difficult to estimate, but what can be said for sure is that it will never exceed the cardinality of the outer stream. It is assumed that half of the records of the outer stream not match the join condition.

```
cardinality = cardinality(outer) * 0.5

cost = cardinality(outer) + cardinality(outer) * cost(inner first row)
```

Here `cost(inner first row)` is the cost of selecting the first record that matches the relation condition. Since the `NOT EXISTS` predicate may contain a complex subquery, rather than reading from a single table, it is not so easy to calculate. It is also quite obvious that for this algorithm to work effectively, the optimizer must switch to the `FIRST ROWS` optimization strategy for executing the inner subquery.

Full Outer Join

This type of join is not performed directly, but instead is decomposed into an equivalent form — first one stream is joined to another using a left outer join, then the streams are swapped and an anti-join is performed, and the results of both joins are combined.

Since full outer join streams are semantically equivalent, the optimizer can swap them depending on which is cheaper.

Example 59. Nested Loop for FULL OUTER JOIN

```
SELECT *
FROM RDB$RELATIONS R
FULL JOIN RDB$RELATION_FIELDS RF ON R.RDB$RELATION_NAME = RF.RDB$RELATION_NAME
```

```
PLAN JOIN (JOIN (RF NATURAL, R INDEX (RDB$INDEX_0)), JOIN (R NATURAL, RF INDEX
(RDB$INDEX_4)))
```

```
[cardinality=3375.0, cost=23980.4]
Select Expression
    [cardinality=3375.0, cost=23980.4]
    -> Full Outer Join
        [cardinality=3200.0, cost=22580.4]
        -> Nested Loop Join (outer)
            [cardinality=2428.0, cost=2428.0]
            -> Table "RDB$RELATION_FIELDS" as "RF" Full Scan
            [cardinality=1.3, cost=4.3]
            -> Filter
                [cardinality=1.3, cost=4.3]
                -> Table "RDB$RELATIONS" as "R" Access By ID
                    -> Bitmap
                        -> Index "RDB$INDEX_0" Unique Scan
        [cardinality=175.0, cost=1400.0]
        -> Nested Loop Join (outer)
            [cardinality=350.0, cost=350.0]
            -> Table "RDB$RELATIONS" as "R" Full Scan
            [cardinality=1.0, cost=4.0]
            -> Filter
                [cardinality=9.1, cost=12.1]
                -> Table "RDB$RELATION_FIELDS" as "RF" Access By ID
                    -> Bitmap
                        -> Index "RDB$INDEX_4" Range Scan (full match)
```

Before Firebird 3.0 it was not possible to use indexes for full outer join, which made this type of join extremely inefficient. The situation could be solved by rewriting the query as follows.

```
SELECT *
FROM RDB$RELATION_FIELDS RF
LEFT JOIN RDB$RELATIONS R ON R.RDB$RELATION_NAME = RF.RDB$RELATION_NAME
UNION ALL
SELECT *
FROM RDB$RELATIONS R
LEFT JOIN RDB$RELATION_FIELDS RF ON R.RDB$RELATION_NAME = RF.RDB$RELATION_NAME
WHERE RF.RDB$RELATION_NAME IS NULL
```

The second part of the query is essentially an anti-join. The optimizer now does such a transformation for you automatically.

Joining with a Stored Procedure

When using inner joins with a stored procedure, there are two options:

- the input parameters of the stored procedure do not depend on other streams;
- the input parameters of the stored procedure depend on the input streams.

These cases are handled differently. If the input parameters of a procedure do not depend on other streams, then the procedure automatically becomes the leading stream. This allows it to be executed once instead of being executed again at each iteration (since we cannot apply an index to it).

Example 60. Joining with an uncorrelated stored procedure

```
SELECT *
FROM SP_PEDIGREE(?) P
JOIN HORSE ON HORSE.CODE_HORSE = P.CODE_HORSE
```

```
PLAN JOIN (P NATURAL, HORSE INDEX (PK_HORSE))
```

```
[cardinality=1000.0, cost=4000.0]
Select Expression
    [cardinality=1000.0, cost=4000.0]
    -> Nested Loop Join (inner)
        [cardinality=1000.0, cost=????]
        -> Procedure "SP_PEDIGREE" as "P" Scan
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "HORSE" Access By ID
                -> Bitmap
                    -> Index "PK_HORSE" Unique Scan
```

If the input parameters of a procedure depend on other streams, the optimizer determines the presence of these dependencies and sorts the streams in such a way that the procedure becomes slaved to the input streams.

Example 61. Join with a correlated stored procedure

```
SELECT *
FROM
  HORSE
  CROSS JOIN SP_PEDIGREE(HORSE.CODE_HORSE) P
WHERE HORSE.CODE_COLOR = ?
```

```
PLAN JOIN (HORSE INDEX (FK_HORSE_COLOR), P NATURAL)
```

```
[cardinality=2377700.0, cost=2618.7]
Select Expression
    [cardinality=2377700.0, cost=2618.7]
    -> Nested Loop Join (inner)
        [cardinality=2377.7, cost=2618.7]
        -> Filter
            [cardinality=2377.7, cost=2618.7]
            -> Table "HORSE" Access By ID
                -> Bitmap
                    -> Index "FK_HORSE_COLOR" Range Scan (full match)
        [cardinality=1000.0, cost=???]
        -> Procedure "SP_PEDIGREE" as "P" Scan
```

Firebird before version 3.0 could not determine the dependence of the procedure's input parameters on other streams. In this case, the procedure was placed forward when no record had been fetched from the table yet. Accordingly, the `isc_no_cur_rec` ("no current record for fetch operation") error occurred at the execution stage. To bypass this problem, an explicit indication of the join order was used using the syntax:

```
SELECT *
FROM
  HORSE
  LEFT JOIN SP_PEDIGREE(HORSE.CODE_HORSE) P ON 1=1
WHERE HORSE.CODE_COLOR = ?
```

In this case, the table will always be read before the procedure and everything will work correctly.

Joining with table expressions

Table expressions are subqueries that are used as data sources in the main query. There are two types of table expressions:

- derived tables;
- common table expressions (CTE).

A derived table is a table expression that appears in the FROM clause of a query. Derived tables are a query in parentheses. You can give this query an alias, after which the table expression can be treated as a regular table (used as a data source).

Example 62. Query using derived table

```
SELECT
  F.RDB$RELATION_NAME
FROM (
    SELECT
      DISTINCT
      RF.RDB$RELATION_NAME,
      RF.RDB$SYSTEM_FLAG
    FROM
      RDB$RELATION_FIELDS RF
    WHERE RF.RDB$FIELD_NAME STARTING WITH 'RDB$'
  ) F
WHERE F.RDB$SYSTEM_FLAG = 1
```

A common table expression (CTE) is a named table expression declared in the WITH clause. A common table expression can be treated as a regular table (used as a data source) just like a derived table. Common table expressions declared below can use table expressions declared above in their body. Common table expressions are divided into recursive and non-recursive. We will talk about recursive CTEs later when considering the corresponding access method. Non-recursive table expressions have the following form:

```
WITH
  <cte_1>, <cte_2>, ... <cte_N>
<main_select_expr>

<cte> ::= _cte_name_ [(<column_aliases>)] AS <select_expr>
```

The example above can be rewritten using CTE as follows:

Example 63. Query using CTE

```
WITH
  F AS (
    SELECT
      DISTINCT
      RF.RDB$RELATION_NAME,
      RF.RDB$SYSTEM_FLAG
    FROM
      RDB$RELATION_FIELDS RF
    WHERE RF.RDB$FIELD_NAME STARTING WITH 'RDB$'
  )
SELECT
  F.RDB$RELATION_NAME
FROM F
WHERE F.RDB$SYSTEM_FLAG = 1
```

How will joins behave when using table expressions? This will depend on the contents of the table expression. In the simplest cases, a query with a table expression can be transformed into an equivalent query without using a table expression. In other words, simple queries are expanded to the base tables. In more complex cases, the table expression is always selected by the leading stream in inner joins. Simple table expressions contain only join operations of streams and their filter conditions. Adding sorting, `DISTINCT`, aggregate calculation, grouping, window function calculation, `UNION` and FIRST/SKIP counters turns the table expression into a complex one.

Example 64. Expanding CTEs to base tables

```
WITH
  FIELDS AS (
    SELECT
      RF.RDB$RELATION_NAME,
      RF.RDB$FIELD_NAME,
      F.RDB$FIELD_TYPE
    FROM
      RDB$RELATION_FIELDS RF
      JOIN RDB$FIELDS F ON F.RDB$FIELD_NAME = RF.RDB$FIELD_SOURCE
  )
SELECT
  R.RDB$RELATION_NAME,
  FIELDS.RDB$FIELD_NAME,
  FIELDS.RDB$FIELD_TYPE
FROM
  RDB$RELATIONS R
  JOIN FIELDS ON FIELDS.RDB$RELATION_NAME = R.RDB$RELATION_NAME
```

```
PLAN JOIN (R NATURAL, FIELDS RF INDEX (RDB$INDEX_4), FIELDS F INDEX (RDB$INDEX_2))
```

```
[cardinality=2182.9, cost=4858.3]
Select Expression
    [cardinality=2182.9, cost=4858.3]
    -> Nested Loop Join (inner)
        [cardinality=233.7, cost=233.7]
        -> Table "RDB$RELATIONS" as "R" Full Scan
        [cardinality=9.34, cost=12.34]
        -> Filter
            [cardinality=9.34, cost=12.34]
            -> Table "RDB$RELATION_FIELDS" as "FIELDS RF" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_4" Range Scan (full match)
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "RDB$FIELDS" as "FIELDS F" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_2" Unique Scan
```

In this case, the CTE `FIELDS` was expanded to the base tables and the optimization was as if we were simply joining the tables inside the CTE to `RDB$RELATIONS R`. Now let's add sorting inside the CTE (it does not change the cardinality).

Example 65. Complex CTEs that cannot be expanded to base tables

```
WITH
  FIELDS AS (
    SELECT
      RF.RDB$RELATION_NAME,
      RF.RDB$FIELD_NAME,
      F.RDB$FIELD_TYPE
    FROM
      RDB$RELATION_FIELDS RF
      JOIN RDB$FIELDS F ON F.RDB$FIELD_NAME = RF.RDB$FIELD_SOURCE
    ORDER BY F.RDB$FIELD_TYPE
  )
SELECT
  R.RDB$RELATION_NAME,
  FIELDS.RDB$FIELD_NAME,
  FIELDS.RDB$FIELD_TYPE
FROM
  RDB$RELATIONS R
  JOIN FIELDS ON FIELDS.RDB$RELATION_NAME = R.RDB$RELATION_NAME
```

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

```
Select Expression
    [cardinality=2428.4, cost=21855.6]
    -> Nested Loop Join (inner)
        [cardinality=2428.4, cost=12142.0]
        -> Refetch
            [cardinality=2428.4, cost=12142.0]
            -> Sort (record length: 44, key length: 8)
                [cardinality=2428.4, cost=12142.0]
                -> Nested Loop Join (inner)
                    [cardinality=2428.4, cost=2428.4]
                    -> Table "RDB$RELATION_FIELDS" as "FIELDS RF" Full Scan
                    [cardinality=1.0, cost=4.0]
                    -> Filter
                        [cardinality=1.0, cost=4.0]
                        -> Table "RDB$FIELDS" as "FIELDS F" Access By ID
                            -> Bitmap
                                -> Index "RDB$INDEX_2" Unique Scan
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "RDB$RELATIONS" as "R" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_0" Unique Scan
```

Here the plan changed completely. First the query inside the CTE was executed, and only then the other streams joined it.

This behavior may change in the future, so don't expect complex CTEs to always be the leading thread in inner joins.

If you are not satisfied with this join order, you can always use the join order specification via `LEFT JOIN` followed by `IS NOT NULL` filtering.

Example 66. Complex CTEs joined by `LEFT JOIN`

```
WITH
  FIELDS AS (
    SELECT
      RF.RDB$RELATION_NAME,
      RF.RDB$FIELD_NAME,
      F.RDB$FIELD_TYPE
    FROM
      RDB$RELATION_FIELDS RF
      JOIN RDB$FIELDS F ON F.RDB$FIELD_NAME = RF.RDB$FIELD_SOURCE
    ORDER BY F.RDB$FIELD_TYPE
  )
SELECT
  R.RDB$RELATION_NAME,
  FIELDS.RDB$FIELD_NAME,
  FIELDS.RDB$FIELD_TYPE
FROM
  RDB$RELATIONS R
  LEFT JOIN FIELDS ON FIELDS.RDB$RELATION_NAME = R.RDB$RELATION_NAME
WHERE FIELDS.RDB$RELATION_NAME IS NOT NULL
```

```
PLAN JOIN (R NATURAL, SORT (JOIN (FIELDS RF INDEX (RDB$INDEX_4), FIELDS F INDEX
(RDB$INDEX_2))))
```

```
[cardinality=1091.45, cost=12082.3]
Select Expression
    [cardinality=1091.45, cost=12082.3]
    -> Filter
        [cardinality=2182.9, cost=12082.3]
        -> Nested Loop Join (outer)
            [cardinality=233.7, cost=233.7]
            -> Table "RDB$RELATIONS" as "R" Full Scan
            [cardinality=9.34, cost=50.7]
            -> Refetch
                [cardinality=9.34, cost=50.7]
                -> Sort (record length: 44, key length: 8)
                    [cardinality=9.34, cost=50.7]
                    -> Nested Loop Join (inner)
                        [cardinality=9.34, cost=12.34]
                        -> Filter
                            [cardinality=9.34, cost=12.34]
                            -> Table "RDB$RELATION_FIELDS" as "FIELDS RF" Access By ID
                                -> Bitmap
                                    -> Index "RDB$INDEX_4" Range Scan (full match)
                        [cardinality=1.0, cost=4.0]
                        -> Filter
                            [cardinality=1.0, cost=4.0]
                            -> Table "RDB$FIELDS" as "FIELDS F" Access By ID
                                -> Bitmap
                                    -> Index "RDB$INDEX_2" Unique Scan
```

In addition to derived tables independent of outer streams, the standard provides for derived tables dependent on outer streams. To be able to use such derived tables, the keyword `LATERAL` must be used before the table expression. Common table expressions (CTE) dependent on outer streams are not possible.

For LATERAL derived tables, the following join types make sense: `CROSS JOIN` and `LEFT JOIN`. The syntax allows other join types as well. A feature of joins with lateral derived tables is that they can only be performed by the nested loop join algorithm. Another feature concerns the execution of `CROSS JOIN`. The fact is that due to the fact that a LATERAL derived table depends on external streams, it cannot be set as the leading stream in joins. The optimizer recognizes the presence of dependencies and selects the correct join order.

Example 67. CROSS JOIN LATERAL

```
SELECT
  R.RDB$RELATION_NAME,
  FIELDS.RDB$FIELD_NAME
FROM
  RDB$RELATIONS R
  CROSS JOIN LATERAL (
    SELECT RF.RDB$FIELD_NAME
    FROM RDB$RELATION_FIELDS RF
    WHERE RF.RDB$RELATION_NAME = R.RDB$RELATION_NAME
    ORDER BY RF.RDB$FIELD_POSITION
    FETCH FIRST ROW ONLY
  ) FIELDS
```

```
PLAN JOIN (R NATURAL, SORT (FIELDS RF INDEX (RDB$INDEX_4)))
```

```
[cardinality=233.7, cost=3351.26]
Select Expression
    [cardinality=233.7, cost=3351.26]
    -> Nested Loop Join (inner)
        [cardinality=233.7, cost=233.7]
        -> Table "RDB$RELATIONS" as "R" Full Scan
        [cardinality=1.0, cost=12.34]
        -> First N Records
            [cardinality=9.34, cost=12.34]
            -> Refetch
                [cardinality=9.34, cost=12.34]
                -> Sort (record length: 28, key length: 8)
                    [cardinality=9.34, cost=12.34]
                    -> Filter
                        [cardinality=9.34, cost=12.34]
                        -> Table "RDB$RELATION_FIELDS" as "FIELDS RF" Access By ID
                            -> Bitmap
                                -> Index "RDB$INDEX_4" Range Scan (full match)
```

Join with views

View is a virtual (logical) table that represents a named query (synonym for a query) that will be substituted as a table expression when using the view.

From the optimizer's perspective, joins to views are no different from joins to table expressions described above. The only significant difference is that, unlike derived tables, views cannot be correlated.

4.1.2. Hash join

Hash join is an alternative algorithm for performing a join. Joining streams using the HASH JOIN algorithm has been available since Firebird 3.0.

When joining by the hashing method, the input streams are always divided into a master and a slave, and the slave is usually the stream with the lowest cardinality. First, the smaller (slave) stream is read entirely into an internal buffer. During reading, a hash function is applied to each link key and the pair {hash, pointer in buffer} is written to the hash table. Then the master stream is read and its link key is tried in the hash table. If a match is found, the records of both streams are joined and output. In the case of several duplicates of a given key in the slave table, several records will be output. If the key does not appear in the hash table, we move on to the next record of the master stream, and so on.

Obviously, replacing key comparison with hash comparison is possible only when comparing for strict equality of keys. Thus, a join with a connection condition of the form (`ON MASTER.F > SLAVE.F`) cannot be performed by the Hash Join algorithm.

However, this method has one advantage over nested loop joins — it allows expression-based hash joins. For example, a join with a link condition like (`ON MASTER.F + 1 = SLAVE.F + 2`) is easily implemented using hash joins. The reason is obvious: there is no dependency between streams and, therefore, no requirement to use indexes.

Collisions may occur during the construction of a hash table. A collision is a situation where the same hash function value is obtained for different key values. Therefore, after the hash function value matches, the connection predicate is always recalculated. Collisions are saved as a list sorted by keys, which makes it possible to perform a binary search along the collision chain.

In the current implementation, the hash table has a fixed size of 1009 slots, it does not change depending on the cardinality of the hashed stream. Also, the hash table size does not increase and rehashing does not occur when the length of collision chains is exceeded. This means that hashing join is effective only with a relatively small cardinality of the slave stream. If the cardinality of the hashed stream exceeds 1009000 records, then other join algorithms are selected (in the presence of indexes, NESTED LOOP JOIN; in the absence, MERGE JOIN).

This may change in future versions of Firebird.

Why 1009000 records? 1009000 / 1009 slots = 1000 possible collisions, searching by sorted collisions takes log2(1000) = 10 steps, which is already considered inefficient.

The hash join algorithm can handle multiple AND join conditions. In this case, the hash function is calculated for multiple fields included in the join condition (or multiple links if more than one stream is joined). However, in the case of OR join conditions, the hash join method cannot be applied. It should be noted here that the hash function is not always calculated for all fields included in the join condition, it can be calculated only for some of them. This happens if the number of link keys exceeds 8. In this case, if the hash function value matches, more records than necessary will be returned from the hash table. The extra records will be filtered out after all predicates in the join condition have been calculated. In this case, the efficiency of hash join decreases. The Explain plan does not display the number of keys involved in the hash function (this information was added in Firebird 6.0).

Another feature of hash join is that it can be performed when comparing keys for strict equality taking into account `NULL` values. That is, both `=` and `IS NOT DISTINCT FROM` are allowed. At the same time, no difference is made between these predicates when constructing a hash table, although it is obvious that if the `=` predicate is used, then records with the `NULL` key can be omitted from the hash table. This leads to increased memory consumption and decreased performance of hash join if there are many keys with the `NULL` value. This drawback is planned to be fixed in future versions of Firebird (see CORE-7769).

Before Firebird 5.0, the HASH JOIN join method was used only when there were no indexes by the join condition or they were not applicable, otherwise the optimizer chose the NESTED LOOP join algorithm using indexes. In fact, this is not always optimal. If a large stream joins a small table by the primary key, then each record of such a table will be read many times, and the index pages, if used, will also be read many times. When using a HASH JOIN join, the smaller table will be read exactly once. Naturally, the cost of hashing and probing is not free, so the choice of which algorithm to use is based on cost.

The cost of joining using the HASH JOIN method consists of the following parts:

- the cost of extracting records from the data stream for hashing;
- the cost of copying records into the hash table (including the cost of calculating the hash function);
- the cost of probing the hash table and the cost of copying for records for which the hashes matched.

The following formulas are used to estimate cardinality and cost (see InnerJoin.cpp):

```
// Cardinality of the output stream
cardinality = outerCardinality * innerCardinality * linkSelectivity

// Cardinality of a hash table if not all fields of the relation condition are keys of the hash table
hashCardinality = innerCardinality * outerCardinality * hashSelectivity

// A hash table is cardinal if all fields of the relation condition are keys of the hash table
hashCardinality = innerCardinality

cost =
    // hashed stream retrieval
    innerCost +
    // hashing cost
    hashCardinality * (COST_FACTOR_MEMCOPY + COST_FACTOR_HASHING) +
    // probing + copying cost
    outerCardinality * (COST_FACTOR_HASHING + innerCardinality * linkSelectivity * COST_FACTOR_MEMCOPY);

COST_FACTOR_MEMCOPY = 0.5

COST_FACTOR_HASHING = 0.5
```

Where

- `hashSelectivity` — selectivity of the link keys by which the hash function was calculated;
- `linkSelectivity` — selectivity of the link condition;
- `innerCost` — cost of retrieving records for the hashed stream;
- `innerCardinality` — cardinality of the slave (hashed) stream;
- `outerCardinality` — cardinality of the leading stream;
- `COST_FACTOR_MEMCOPY` - cost of copying a record from/to memory;
- `COST_FACTOR_HASHING` - cost of calculating the hash function.

В Legacy плане выполнения соединение хешированием отображается словом "HASH", за которым в скобках через запятую описываются входные потоки.

In the Explain of the execution plan, the hash join is shown as a tree with the root labeled “Hash Join” followed by the join type in parentheses. The joinable streams are described at the level below. The slave table hash is shown as “Record Buffer” followed by the record length in parentheses as (`record length: <length>`) (see Record Buffer).

Hash Join (inner)

Example 68. Hash Join (inner)

```
SELECT *
FROM
  RDB$RELATIONS R
  JOIN RDB$PAGES P ON P.RDB$RELATION_ID = R.RDB$RELATION_ID
```

```
PLAN HASH (P NATURAL, R NATURAL)
```

```
[cardinality=829.7, cost=1369.0]
Select Expression
    [cardinality=829.7, cost=1369.0]
    -> Filter
        [cardinality=829.7, cost=1369.0]
        -> Hash Join (inner)
            [cardinality=511.25, cost=511.25]
            -> Table "RDB$PAGES" as "P" Full Scan
            [cardinality=350.6, cost=701.2]
            -> Record Buffer (record length: 1345)
                [cardinality=350.6, cost=350.6]
                -> Table "RDB$RELATIONS" as "R" Full Scan
```

Hash Join (outer)

Currently, one-way outer join by hashing method is not implemented in Firebird. Its execution algorithm is similar to inner join execution, with one exception - the inner (slave or optional) stream is always hashed. For each record of the outer (mandatory) stream, the hash table is tested, if a match is found, then the two records are merged into one and output. If no match is found, then the record from the outer stream with the required number of NULL values is output.

Hash Join (semi)

This data access method can be used to execute subqueries in `EXIST` and `IN (<select>)` predicates, as well as other predicates that are converted to `EXIST` (`ANY`, `SOME`). It is available since Firebird 5.0.1. By default, this feature is disabled, to enable it, you must set the `SubQueryConversion` configuration parameter to `true` in the `firebird.conf` or `database.conf` file.

Not every subquery in `EXIST` can be converted to a semi-join. If the subquery contains FETCH/FIRST/SKIP/ROWS constraints, then the subquery cannot be converted to a semi-join and will be executed as a normal correlated subquery. As with an inner join, to perform a semi-join using the hash method, it is necessary that the keys are compared for strict equality, expressions are allowed in the join conditions, and the join conditions can be combined using `AND`.

The algorithm of the semi-join by the hash method is as follows. The subquery stream is selected as the slave stream, which is read entirely into the internal buffer. During the reading process, a hash function is applied to each link key and the pair {hash, pointer in the buffer} is written to the hash table. After that, the leading stream is read and its link key is tested in the hash table. If a match is

found, the record from the outer stream is output. If the key does not appear in the hash table, we move on to the next record of the leading stream, and so on.

The cardinality of the output stream is very difficult to estimate, but what can be said for sure is that it will never exceed the cardinality of the outer stream. It is assumed that half of the records of the outer stream match the join condition.

Currently, the cost of a hashing semi-join is not calculated.

```
cardinality = cardinality(outer) * 0.5
```

Example 69. Hash Join (semi)

```
SELECT *
FROM RDB$RELATIONS R
WHERE EXISTS (
  SELECT *
  FROM RDB$PAGES P
  WHERE P.RDB$RELATION_ID = R.RDB$RELATION_ID
    AND P.RDB$PAGE_SEQUENCE > 5
)
```

```
PLAN HASH (R NATURAL, P NATURAL)
```

```
[cardinality=175.3, cost=1548.4]
Select Expression
    [cardinality=175.3, cost=1548.4]
    -> Filter
        [cardinality=175.3, cost=1548.4]
        -> Hash Join (semi)
            [cardinality=350.6, cost=350.6]
            -> Table "RDB$RELATIONS" as "R" Full Scan
            [cardinality=255.625, cost=1022.5]
            -> Record Buffer (record length: 33)
                [cardinality=255.625, cost=511.25]
                -> Filter
                    [cardinality=511.25, cost=511.25]
                    -> Table "RDB$PAGES" as "P" Full Scan
```

In this case, the relation condition is `P.RDB$RELATION_ID = R.RDB$RELATION_ID`. This query is "sort of" transformed into the following form:

```
SELECT *
FROM
  RDB$RELATIONS R
  SEMI JOIN (
    SELECT *
    FROM RDB$PAGES
    WHERE RDB$PAGES.RDB$PAGE_SEQUENCE > 5
  ) P ON P.RDB$RELATION_ID = R.RDB$RELATION_ID
```

Of course, such syntax does not exist in SQL. It is demonstrated for better understanding of what is happening in terms of joins.

Hash Join (anti)

This access method can be used to execute subqueries in the `NOT EXIST` predicate, as well as other predicates that are converted to `NOT EXIST (ALL)`. In addition, some one-way outer queries can also be converted to it under certain conditions. It is not implemented in current versions of Firebird.

The algorithm of anti-join by the hashing method is as follows. The subquery stream is selected as a slave stream, which is read entirely into the internal buffer. During the reading process, a hash function is applied to each link key and the pair {hash, pointer in the buffer} is written to the hash table. After that, the leading stream is read and its link key is tested in the hash table. If a match is not found, then the record from the outer stream is output. If a match is found, then we move on to the next record of the leading stream, and so on.

4.1.3. One-pass merge

Merge is an alternative algorithm for implementing a join. In this case, the input streams are completely independent. For the operation to be performed correctly, the streams must be pre-ordered by the join key, after which a binary merge tree is built. Then, during fetch, records are read from both streams and their link keys are compared for equality. If the keys match, the record is returned to the output. Then new records are read from the input streams and the process is repeated. Merge is always performed in one pass, i.e. each input stream is read only once. This is possible due to the ordered arrangement of the link keys in the input streams.

However, this algorithm cannot be used for all types of joins. As noted above, a strict equality comparison of keys is required. Thus, a join with a join condition of the form (`ON MASTER.F > SLAVE.F`) cannot be performed using a one-pass merge. To be fair, it should be noted that joins of this type cannot be efficiently performed in any way, since even in the case of using the nested loop join algorithm, there will be repeated reads from the internal stream at each iteration.

In some DBMS, for example Oracle, it is possible to join using the merge method not only for strict equality, that is, joins with join conditions of the type (`ON MASTER.F > SLAVE.F`) are possible.

However, this method has one advantage over the nested loop join algorithm — it allows expression-based merging. For example, a join with a link condition like (`ON MASTER.F + 1 = SLAVE.F + 2`) is easily performed by the merge method. The reason is clear: there is no dependency between threads and, therefore, no requirement to use indexes.

An attentive reader might have noticed that the description of the merge algorithm does not specify the mode of operation of the method in the case of flow dependencies (one-way outer join). The answer is simple: this algorithm is not supported for such joins. In addition, it is also not supported for full outer joins, semi-joins, and anti-joins.

Support for the outer joins algorithm is planned for future Firebird versions.

This algorithm can handle multiple link conditions combined via AND. In this case, the input streams are simply sorted by several fields. However, in the case of combining link conditions via OR, the merge method cannot be applied.

The main disadvantage of this algorithm is the requirement to sort both streams by join keys. If the streams are already sorted, then this join algorithm is the cheapest among others (NESTED LOOP, HASH JOIN). However, usually the streams to be joined are not sorted by join keys, and therefore they have to be sorted, which dramatically increases the cost of performing the join by this method. Unfortunately, at the moment the optimizer cannot determine that the streams are already sorted by join keys, and therefore this algorithm may not be used in cases where it would really be cheaper.

This shortcoming is planned to be corrected in future versions of Firebird.

Before Firebird 3.0, merge joins were used only when the nested loop join algorithm was impossible or suboptimal, i.e. primarily when there were no indexes by the link condition or they were not applicable, and when there was no dependency between the input streams. Starting with Firebird 3.0, the merge join algorithm was disabled, and hash join was always used instead. Starting with Firebird 5.0, merge join became available again. It is used only when the streams being joined are too large and hash join becomes ineffective (the cardinality of all streams being joined is greater than 1009000 records, see Hash join), and also when there are no indexes by the link condition, which makes the nested loop join algorithm suboptimal.

The following formulas are used to estimate the cost and cardinality of a join:

```
cost = cost_1 + cost_2

cardinality = cardinality_1 * cardinality_2 * selectivity(link)
```

Where `cost_1`, `cost_2` is the cost of extracting data from sorted input streams. And since the cost of external sorting is not calculated, the cost of merge join cannot be estimated correctly.

In a Legacy execution plan, a merge join is represented by the word "MERGE" followed by parentheses and a comma-separated list of input streams.

In the Explain execution plan, the merge join is displayed as a tree, the root of which is labeled “Merge Join” followed by the join type in parentheses. The level below describes the streams being joined, which are sorted by the outer sort.

Example 70. Merge Join (inner)

```
SELECT *
FROM
  WORD_DICTIONARY WD1
  JOIN WORD_DICTIONARY WD2 ON WD1.PARAMS = WD2.PARAMS
```

```
PLAN MERGE (SORT (WD1 NATURAL), SORT (WD2 NATURAL))
```

```
[cardinality=22307569204, cost=???]
Select Expression
    [cardinality=22307569204, cost=???]
    -> Filter
        [cardinality=22307569204, cost=???]
        -> Merge Join (inner)
            [cardinality=4723000, cost=???]
            -> Sort (record length: 710, key length: 328)
                [cardinality=4723000, cost=???]
                -> Table "WORD_DICTIONARY" as "WD1" Full Scan
            [cardinality=4723000, cost=???]
            -> Sort (record length: 710, key length: 328)
                [cardinality=4723000, cost=4723000]
                -> Table "WORD_DICTIONARY" as "WD2" Full Scan
```

The last example is artificial because it is quite difficult to force the optimizer to perform a merge join.

4.2. Union

The name of the access method speaks for itself. This access method performs a SQL union operation. There are two modes of execution of this operation: `ALL` and `DISTINCT`. In the first case, the implementation is trivial: this method simply reads the first input stream and outputs it, upon receiving EOF from it, it starts reading the second input stream, and so on. In the `DISTINCT` case, it is necessary to eliminate complete duplicates of records present as a result of the union. To do this, a sorting filter is placed at the output of the union method, operating in the “truncating” mode for all fields.

The cost of executing a union is equal to the total cost of all input streams, the cardinality is also obtained by summing. In the `DISTINCT` mode, the resulting cardinality is divided by 10.

In the Legacy execution plan, the union is displayed as separate plans for each of the input streams.

In the Explain execution plan, the union is displayed as a tree, the root of which is labeled as “Union”. The streams being merged are described at the level below.

Example 71. UNION ALL

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

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

```
[cardinality=90.33, cost=873.3]
Select Expression
    [cardinality=90.33, cost=873.3]
    -> Union
        [cardinality=35.06, cost=350.6]
        -> Filter
            [cardinality=350.6, cost=350.6]
            -> Table "RDB$RELATIONS" Full Scan
        [cardinality=55.27, cost=552.7]
        -> Filter
            [cardinality=552.7, cost=552.7]
            -> Table "RDB$PROCEDURES" Full Scan
```

Example 72. UNION DISTINCT

```
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 SORT (RDB$RELATIONS NATURAL, RDB$PROCEDURES NATURAL)
```

```
[cardinality=9.033, cost=873.3]
Select Expression
    [cardinality=9.033, cost=873.3]
    -> Unique Sort (record length: 44, key length: 8)
        [cardinality=90.33, cost=873.3]
        -> Union
            [cardinality=35.06, cost=350.6]
            -> Filter
                [cardinality=350.6, cost=350.6]
                -> Table "RDB$RELATIONS" Full Scan
            [cardinality=55.27, cost=552.7]
            -> Filter
                [cardinality=552.7, cost=552.7]
                -> Table "RDB$PROCEDURES" Full Scan
```

4.2.1. Materialization of non-deterministic expressions

Table expressions in Firebird have one nasty feature, namely when accessing table expression columns that reference expressions, these expressions are evaluated each time the table expression column is mentioned. This is easily demonstrated when using non-deterministic functions.

Try running the following query:

```
WITH
  T AS (
    SELECT GEN_UUID() AS UUID
    FROM RDB$DATABASE
  )
SELECT
  UUID_TO_CHAR(UUID) AS ID1,
  UUID_TO_CHAR(UUID) AS ID2
FROM T
```

The result will be something like

```
ID1                                  ID2
====================================    ====================================
3C8CA94D-9D05-4A49-8788-1D9024193C4E    D5E86F5C-BF48-4AA3-AB6D-3EF9C9B58B52
```

Although you expected the values in the columns to be the same.

This also occurs when sorting by column reference. For example:

```
SELECT GEN_ID(SEQ_SOME, 1) AS ID
FROM RDB$DATABASE
ORDER BY 1
```

The sequence will increase not by 1, as you expected, but by 2.

There is one trick that allows you to “materialize” the results of expression calculations in table expressions. To do this, it is enough to make a `UNION` with a query returning 0 records, naturally the total number of columns of both queries must be the same. Let’s try to rewrite the first query as follows:

```
WITH
  T AS (
    SELECT GEN_UUID() AS UUID
    FROM RDB$DATABASE
    UNION ALL
    SELECT NULL FROM RDB$DATABASE WHERE FALSE
  )
SELECT
  UUID_TO_CHAR(UUID) AS ID1,
  UUID_TO_CHAR(UUID) AS ID2
FROM T
```

Now the values of ID1 and ID2 will be the same.

```
ID1                                  ID2
====================================    ====================================
9599C281-DD96-4CC7-9C05-6259CCAB467F    9599C281-DD96-4CC7-9C05-6259CCAB467F
```

4.2.2. Materialization of subqueries

What happens if a subquery is used as a column in a table expression? A subquery can be quite heavy to execute repeatedly. Fortunately, in this case, the optimizer does all the work for us. Try executing the following query:

```
WITH T
  AS (
    SELECT
      (SELECT COUNT(*)
       FROM RDB$RELATION_FIELDS RF
       WHERE RF.RDB$RELATION_NAME = R.RDB$RELATION_NAME) AS CNT
    FROM RDB$RELATIONS R
  )
SELECT
  SUM(T.CNT) AS CNT,
  SUM(T.CNT) * 1e0 / COUNT(*) AS AVG_CNT
FROM T
```

Despite the repeated mention of `T.CNT`, the subquery will be executed once for each record in `RDB$RELATIONS`.

In the Explain plan you will see the following:

```
Sub-query
    -> Singularity Check
        -> Aggregate
            -> Filter
                -> Table "RDB$RELATION_FIELDS" as "T RF" Access By ID
                    -> Bitmap
                        -> Index "RDB$INDEX_4" Range Scan (full match)
Select Expression
    -> Aggregate
        -> Materialize
            -> Table "RDB$RELATIONS" as "T R" Full Scan
```

Here “Materialize” means materialization of the subquery results for each record from `RDB$RELATIONS`. This is not a separate access method, but an inner union (UNION) inside a table expression with an empty set. That is, the same thing happened as described above about the materialization of non-deterministic expressions.

4.3. Recursion

This data access method is used to execute recursive table expressions (recursive CTEs). The body of a recursive CTE is a query with `UNION ALL` that combines one or more subqueries called anchors. In addition to the anchors, there are one or more recursive subqueries called recursive elements. These recursive subqueries refer to the recursive CTE itself. So, we have one or more anchored subqueries and one or more recursive subqueries, combined by `UNION ALL`.

The essence of recursion is simple — first, non-recursive subqueries are executed and combined, and for each record of the non-recursive part, the data set is supplemented with records from the recursive part, which can use the result obtained in the previous step. Recursion stops when all recursive parts return no records.

There is another peculiarity when executing recursive CTEs - no predicates from the outer query

can be pushed inside the CTE.

The cardinality and cost of recursion are very difficult to estimate. The optimizer considers the cardinality to be the sum of the cardinalities of the non-recursive parts, which is multiplied by the cardinality added by the joins in the recursive part. The cost is summed up from the cost of selecting all the records of the recursive and non-recursive parts. It is worth noting that such an estimate is sometimes far from the truth.

In the Legacy execution plan, recursion is not displayed.

In the Explain execution plan, recursion is displayed as a tree with the root labeled “Recursion”. The lower level describes the flows being combined. A call to the recursive CTE itself within the CTE is not described as a separate data flow.

Example 73. The simplest recursive query

```
WITH RECURSIVE
  R(N) AS (
    SELECT 1 FROM RDB$DATABASE
    UNION ALL
    SELECT R.N + 1 FROM R
  )
SELECT N FROM R
```

```
PLAN (R RDB$DATABASE NATURAL, )
```

```
[cardinality=1.0, cost=1.0]
Select Expression
    [cardinality=1.0, cost=1.0]
    -> Recursion
        [cardinality=1.0, cost=1.0]
        -> Table "RDB$DATABASE" as "R RDB$DATABASE" Full Scan
```

The query described above has one significant drawback. It makes “infinite” recursion. But this is in theory, and in practice Firebird limits the recursion depth to 1024. Let’s fix this.

Example 74. Recursive query with depth limit

```
WITH RECURSIVE
  R(N) AS (
    SELECT 1 FROM RDB$DATABASE
    UNION ALL
    SELECT R.N + 1 FROM R WHERE R.N < 10
  )
SELECT N FROM R
```

```
PLAN (R RDB$DATABASE NATURAL, )
```

```
[cardinality=1.0, cost=1.0]
Select Expression
    [cardinality=1.0, cost=1.0]
    -> Recursion
        [cardinality=1.0, cost=1.0]
        -> Table "RDB$DATABASE" as "R RDB$DATABASE" Full Scan
        [cardinality=1.0, cost=1.0]
        -> Filter (preliminary)
```

Here in the explain plan you can see the recursive part that is filtered, but the stream itself is not shown. Since the recursive member does not join to other data sources, the cardinality and cost are taken from the non-recursive part as well. Here the cardinality estimate is 1, although in reality 10 records will be returned.

The recursive part can be more complex and contain joins (only inner) to other data sources.

Example 75. Recursive query with join on non-unique index

```
WITH RECURSIVE
  R AS (
    SELECT
      DEPT_NO,
      DEPARTMENT,
      HEAD_DEPT
    FROM DEPARTMENT
    WHERE HEAD_DEPT IS NULL
    UNION ALL
    SELECT
      DEPARTMENT.DEPT_NO,
      DEPARTMENT.DEPARTMENT,
      DEPARTMENT.HEAD_DEPT
    FROM R JOIN DEPARTMENT ON DEPARTMENT.HEAD_DEPT = R.DEPT_NO
  )
SELECT * FROM R
```

```
PLAN (R DEPARTMENT INDEX (RDB$FOREIGN6), R DEPARTMENT INDEX (RDB$FOREIGN6))
```

```
[cardinality=6.890625, cost=20.390625]
Select Expression
    [cardinality=6.890625, cost=20.390625]
    -> Recursion
        [cardinality=2.625, cost=5.625]
        -> Filter
            [cardinality=2.625, cost=5.625]
            -> Table "DEPARTMENT" as "R DEPARTMENT" Access By ID
                -> Bitmap
                    -> Index "RDB$FOREIGN6" Range Scan (full match)
        [cardinality=2.625, cost=5.625]
        -> Filter
            [cardinality=2.625, cost=5.625]
            -> Table "DEPARTMENT" as "R DEPARTMENT" Access By ID
                -> Bitmap
                    -> Index "RDB$FOREIGN6" Range Scan (full match)
```

Here, the recursive part of the query joins the `DEPARTMENT` table on a non-unique index, which is why the cardinalities are multiplied (for each record from the non-recursive member, `2,625` records from the recursive member are joined). In fact, 21 records will be selected.

Let's try to reverse the recursion.

Example 76. Recursive query with join on unique index

```
WITH RECURSIVE
  R AS (
    SELECT
      DEPT_NO,
      DEPARTMENT,
      HEAD_DEPT
    FROM DEPARTMENT
    WHERE DEPT_NO = '672'
    UNION ALL
    SELECT
      DEPARTMENT.DEPT_NO,
      DEPARTMENT.DEPARTMENT,
      DEPARTMENT.HEAD_DEPT
    FROM R JOIN DEPARTMENT ON DEPARTMENT.DEPT_NO = R.HEAD_DEPT
  )
SELECT * FROM R
```

```
PLAN (R DEPARTMENT INDEX (RDB$PRIMARY5), R DEPARTMENT INDEX (RDB$PRIMARY5))
```

```
[cardinality=1.0, cost=8.0]
Select Expression
    [cardinality=1.0, cost=8.0]
    -> Recursion
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "DEPARTMENT" as "R DEPARTMENT" Access By ID
                -> Bitmap
                    -> Index "RDB$PRIMARY5" Unique Scan
        [cardinality=1.0, cost=4.0]
        -> Filter
            [cardinality=1.0, cost=4.0]
            -> Table "DEPARTMENT" as "R DEPARTMENT" Access By ID
                -> Bitmap
                    -> Index "RDB$PRIMARY5" Unique Scan
```

Here, the recursive part of the query joins the `DEPARTMENT` table using a unique index, so the overall cardinality is not increased (for 1 record from the non-recursive member, 1 record from the recursive member is joined). In practice, 4 records will be returned.

Chapter 5. Optimization strategies

In the process of reviewing Firebird access methods, we found out that data sources can be pipelined and buffered. A pipelined data source produces records as it reads its input streams, while a buffered source must first read all records from its input streams before it can produce the first record to its output.

Most access methods are pipelined. External sorting (SORT) and record buffering (Record Buffer) belong to buffered access methods. In addition, some access methods require the use of buffered data sources. Hash Join requires buffering when building a hash table. Merge Join requires ordering of input streams by join keys, for which external sorting is used. Buffering is also required for calculating window functions (Window).

In Firebird, the optimizer can work in two modes:

- `FIRST ROWS` — the optimizer builds a query plan so as to extract only the first rows of the query as quickly as possible;
- `ALL ROWS` — the optimizer builds a query plan so as to extract all rows of the query as quickly as possible.

In most cases, the `ALL ROWS` optimization strategy is required. However, if you have applications with data grids where only the first rows of the result are displayed and the rest are retrieved as needed, then the `FIRST ROWS` strategy may be preferable because it improves response time.

When using the `FIRST ROWS` strategy, the optimizer tries to replace buffered access methods with alternative pipelined ones, if possible and cheaper in terms of the cost of extracting the first row. That is, Hash/Merge join is replaced with Nested Loop Join, and external sorting (Sort) is replaced with index navigation.

By default, the optimization strategy specified in the `OptimizeForFirstRows` parameter of the `firebird.conf` or `database.conf` configuration file is used. `OptimizeForFirstRows = false` corresponds to the `ALL ROWS` strategy, `OptimizeForFirstRows = true` corresponds to the `FIRST ROWS` strategy.

The optimization strategy can be overridden directly in the SQL query text using the `OPTIMIZE FOR` clause. In addition, using first/skip counters in a query implicitly switches the optimization strategy to `FIRST ROWS`. Let's compare query plans that use different optimization strategies.

Example 77. Query with ALL ROWS optimization strategy

```
SELECT
  R.RDB$RELATION_NAME,
  P.RDB$PAGE_SEQUENCE
FROM
  RDB$RELATIONS R
  JOIN RDB$PAGES P ON P.RDB$RELATION_ID = R.RDB$RELATION_ID
OPTIMIZE FOR ALL ROWS
```

```
PLAN HASH (P NATURAL, R NATURAL)
```

```
Select Expression
    -> Filter
        -> Hash Join (inner)
            -> Table "RDB$PAGES" as "P" Full Scan
            -> Record Buffer (record length: 281)
                -> Table "RDB$RELATIONS" as "R" Full Scan
```

Example 78. Query with FIRST ROWS optimization strategy

```
SELECT
  R.RDB$RELATION_NAME,
  P.RDB$PAGE_SEQUENCE
FROM
  RDB$RELATIONS R
  JOIN RDB$PAGES P ON P.RDB$RELATION_ID = R.RDB$RELATION_ID
OPTIMIZE FOR FIRST ROWS
```

```
PLAN JOIN (P NATURAL, R INDEX (RDB$INDEX_1))
```

```
Select Expression
    -> Nested Loop Join (inner)
        -> Table "RDB$PAGES" as "P" Full Scan
        -> Filter
            -> Table "RDB$RELATIONS" as "R" Access By ID
                -> Bitmap
                    -> Index "RDB$INDEX_1" Range Scan (full match)
```

Chapter 6. Conclusion

Above we have considered all types of operations that participate in the execution of SQL queries. For each algorithm, a detailed description and examples are given, including a detailed query execution tree (Explain plan). It is worth noting that Firebird does not implement many access methods compared to competitors, but this is often explained either by the peculiarities of the architecture or by the comparatively greater efficiency of the existing methods.