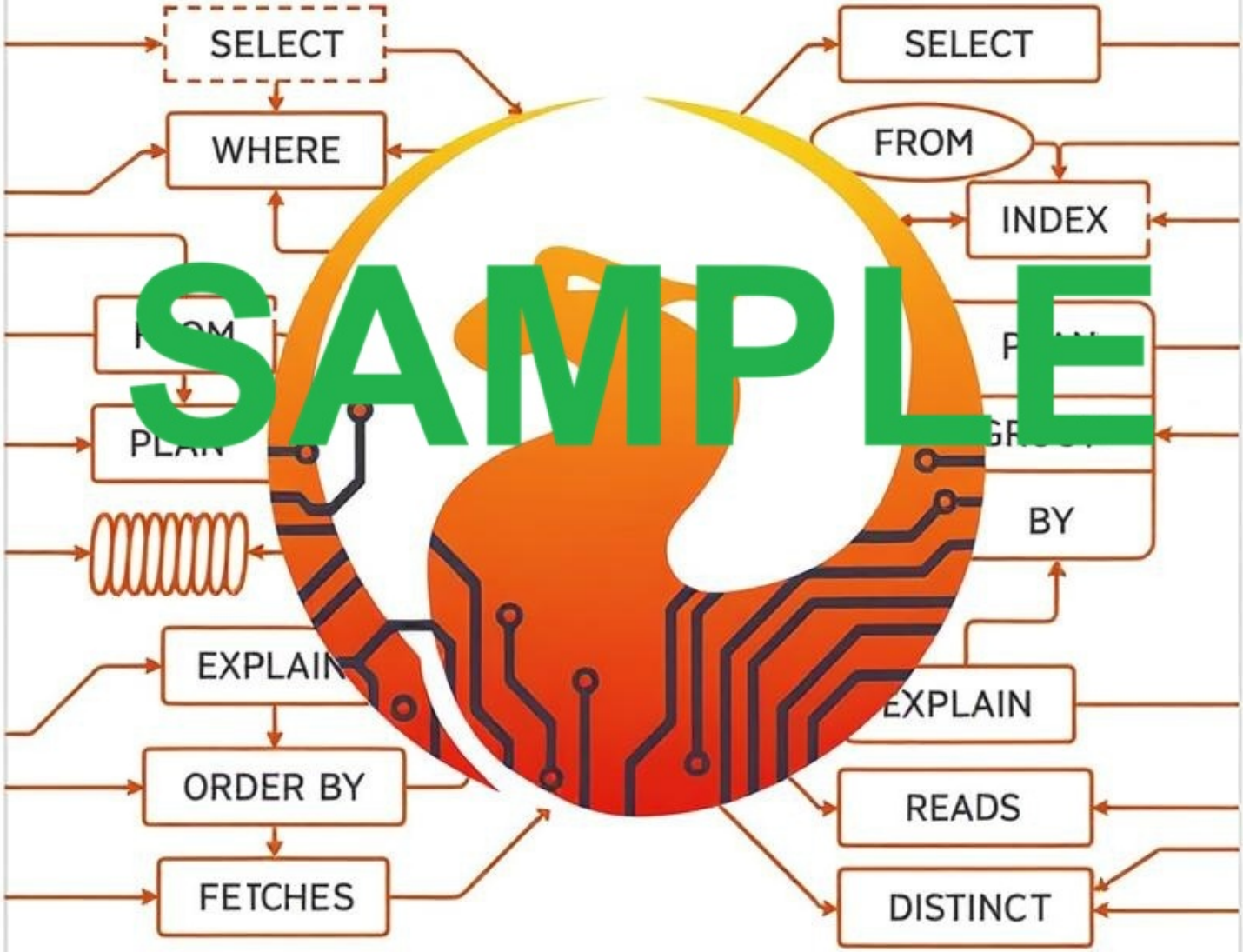
SECRETS OF FIREBIRD SQL OPTIMIZER



D.Simonov

2025

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

Table of Contents

Introduction: The Path to Understanding the Firebird Optimizer

Why Performance Matters

In the modern world of information systems, database performance is not just a technical metric, but a critical factor for business success. A slow system response during data entry irritates users, reduces their productivity, and can lead to loss of customers. Analytical reports that take hours instead of minutes slow down management decision-making. In an era where every second counts, SQL query performance becomes a real battlefield for competitive advantage.

Imagine a typical situation: you have developed an excellent application with a well-thought-out interface, but when trying to open a reference book, the user waits 5-10 seconds. Or a monthly sales report takes hours to generate. A familiar picture? Unfortunately, such problems are widespread, and often the cause lies not in a lack of server resources, but in a lack of understanding of how the database processes queries.

It is especially unpleasant when performance problems manifest themselves as the database size grows: developers use a small database to create SQL queries, on such a volume Firebird returns data very quickly even with a non-optimal query plan, but after several years working with the database becomes painfully slow and slows down more and more, and at some point users abandon the system entirely.

Firebird is a powerful, reliable, and time-tested DBMS that successfully runs on millions of servers worldwide. It can process millions of records efficiently and quickly, but only under one condition: if the developer understands how the query execution mechanism works. This is precisely why this book was written.

The Evolution of Optimization Approaches: From Hardware to Understanding

The history of database optimization is full of misconceptions and myths. In the early days of information systems, when performance left much to be desired, the solution seemed obvious: add more RAM, install a faster processor, switch to SSD drives. This approach can be called "hardware optimization," and it indeed worked… up to a certain point, until it hit the law of diminishing returns.

In the 2000s, when hardware costs began to plummet, many organizations followed the path of increasing capacity. Increasing RAM from 4 to 32 gigabytes gave a noticeable performance boost. Switching to RAID arrays sped up disk operations. But sooner or later, a moment of truth arrived: another doubling of memory no longer brought the expected result. A system with 128 GB of RAM continued to execute some queries painfully slowly, and administrators, looking at system monitors with gigabytes of unused memory, shrugged in bewilderment. The hardware was idle, but the queries crawled.

The next stage is configuration optimization. Developers and administrators study operating system

parameters and Firebird settings: page cache size, sorting parameters, network buffer settings, use the Firebird configuration calculator. This is a step in the right direction, but disappointments lurk here as well. Increasing the cache size from 256 to 131072 pages can either speed up work or, conversely, slow it down (for example, in the Classic architecture). Fine-tuning parameters requires an understanding of how these parameters affect the processing of specific queries.

But the professional community has always known: real optimization requires not only powerful hardware and optimized configuration files. One must know **how the optimizer makes decisions about query execution**.

Of course, we do not deny the importance of hardware resources and correct configuration. Attempting to run a serious production system on a virtual machine with 4 GB of RAM and a slow HDD is doomed to failure. Setting Firebird parameters for a specific task — for example, increasing `TempCacheLimit` for systems with a large number of sorts — can provide a significant boost. But all of these are necessary, but insufficient conditions for high performance.

Breakthrough results can be achieved in only one way: by understanding how the query optimizer works and being able to influence its decisions.

The Trap of Endless Resource Scaling

Let’s consider a typical example illustrating the limitations of the "more hardware — higher performance" approach. A company faced the problem of slow generation of a complex sales report. The first solution seemed obvious: increase the server’s RAM from 16 to 64 GB. They invested money, waited for the reboot, ran the report… and saw 8 minutes instead of the previous 10. There is progress, but disappointment too.

The next step was purchasing expensive latest-generation NVMe SSD drives. Another minute gained — now 7 minutes. Then they switched to a more powerful processor with double the number of cores — another half-minute saved. The hardware budget was exhausted, thousands of dollars were spent in total, but the report still takes 6-7 minutes to generate, which is unacceptable for the company’s business processes.

Now imagine another scenario: a developer who understands the optimizer’s work spent 15 minutes analyzing the query execution plan and discovered that when joining three tables, the optimizer chose a catastrophically non-optimal JOIN order due to the lack of up-to-date statistics. Recalculating statistics with the `SET STATISTICS` command, a slight restructuring of the query using CTE — and the report is generated in 4 seconds. On the old hardware. Without a single ruble of additional investment. A 100x performance increase.

This is not a fictional story. Such situations happen constantly. Queries that take minutes to execute, after proper optimization, start working in fractions of a second. Performance increases of 10, 50, 100, and even more times — this is a reality available to those who understand the mechanisms of the optimizer.

Typical Mistakes: When Good Code Becomes Slow

Why do performance problems arise so often? After all, most developers know SQL, can write

queries, and even use indexes. The problem is that knowing syntax and understanding execution mechanisms are completely different things. It's like knowing traffic rules but not understanding how a car engine works: the car will drive, but at the first serious problem, you will be helpless.

The most common mistakes, even among experienced developers:

Incorrect Indexing — creating redundant indexes or, conversely, lacking necessary indexes where they are critical. A classic example: a developer created an index on a field used in `WHERE`, rejoices in anticipation of acceleration, runs the query... and sees a Full Table Scan in the plan. The optimizer ignores the index due to low selectivity or due to a function in the filter condition. Or the opposite situation: ten indexes are created on one table "just in case," which slows down every insert and update operation, and none of these indexes provides real benefit.

Incorrect JOIN Order — perhaps the most insidious and difficult-to-detect problem. A developer writes a query, joining tables in a logical and intuitively understandable order: first orders, then customers, then products. The query works... slowly. The thing is, the optimizer may choose a completely different order for executing joins, leading to multiple scans of large tables. For example, starting with the products table containing a million records, instead of first filtering the needed orders. Understanding how the optimizer determines the join order and how this can be influenced can literally turn a slow query into a fast one with one line of code.

Ignoring Statistics — the optimizer makes decisions based on statistical data about value distribution in tables. Outdated or missing statistics lead to non-optimal execution plans. Many developers don't even know about the existence of commands for collecting and updating statistics.

Misunderstanding Access Methods — using a full table scan where index access is possible, or conversely, attempting to apply an index to a table with three records. Each access method has its own area of application, and choosing the wrong method can nullify all optimization efforts.

Inefficient Sorts and Groupings — `ORDER BY` and `GROUP BY` operations can be extremely costly if the optimizer is forced to sort large volumes of data. At the same time, there are often ways to avoid sorting or significantly reduce the volume of sorted data.

All these mistakes have one thing in common: they arise not from a lack of knowledge of SQL as a language, but from a lack of understanding of what happens "under the hood" when a query is executed.

Evolution of Firebird and the Optimizer: From Firebird 1 to 5

To truly understand the capabilities of modern Firebird, it is useful to know how the optimizer has evolved over the past two decades. This is not just a historical note — many features of the optimizer's behavior can be explained by knowing its evolutionary path.

Firebird 1.0-1.5: The InterBase Legacy

In the early 2000s, Firebird had just separated from the commercial InterBase and began its journey as an open-source project. The optimizer of that time was relatively simple but reliable. It knew how to use indexes, perform basic joins via Nested Loop Join, and apply simple optimization

rules. Many concepts laid down in those years continue to work today, ensuring continuity and predictability of behavior.

Firebird 2.0-2.5: Expanding Capabilities

This era brought significant improvements. Derived tables appeared, allowing for writing more complex and expressive queries. The optimizer learned to work better with subqueries, applying various transformation techniques. Algorithms for evaluating index selectivity were improved, and new sorting capabilities emerged. It was in version 2.5 that many companies began to seriously use Firebird in large projects, and the optimizer handled this load.

The introduction of Common Table Expressions (CTE) radically changed the approach to writing complex queries. It became possible to break down bulky queries into logical blocks, making them readable, maintainable, and manageable. The optimizer learned to work with recursive CTEs, opening new possibilities for processing hierarchical data — category trees, organizational structures, process routes.

Also, at the Firebird 2.0-2.5 level, statistics for composite index segments appeared, allowing the optimizer to better handle complex conditions.

Firebird 3.0: A Revolutionary Step

Version 3.0 became a real breakthrough and a turning point in Firebird's history.

Window Functions added a powerful toolkit for analytical queries. Tasks like "calculate a running total" or "find the top 3 in each category," which previously required complex self-joins or procedural code with cursors, could now be solved with elegant declarative queries in a few lines. The optimizer received specialized algorithms for efficiently processing window functions, making analytics directly in SQL a reality.

Version 3.0 brought support for Hash Join — an alternative table join algorithm that in certain scenarios works an order of magnitude more efficiently than the traditional Nested Loop Join. For queries joining large tables with good filter selectivity, Hash Join can provide a multiple-fold performance boost. Imagine: a query took 2 minutes with Nested Loop, and after the optimizer chose Hash Join — 5 seconds. Such wonders became reality.

Furthermore, version 3.0 significantly reworked the engine architecture, creating a foundation for further optimizer improvements.

Firebird 4.0: LATERAL and Evolutionary Changes

The appearance of LATERAL joins expanded the horizons of what is possible, allowing the creation of correlated subqueries with access to columns from outer tables. Tasks like "for each customer, find their last 3 orders" or "get top products by categories," which were previously extremely difficult or even impossible within a single efficient SQL query, could now be solved elegantly and quickly.

Also, in Firebird 4.0, optimization of wide sorts using Refetch appeared, expanding optimization possibilities for "wide" queries (reports, exports, etc.).

Firebird 5.0: Modern Capabilities

The latest major version continued the evolution of the optimizer. Additional improvements were made to query cost estimation algorithms, capabilities for subquery transformation were expanded, and new optimizations for specific query patterns were added. The optimizer became "smarter" in choosing between Hash Join and Nested Loop Join, learned to work better with complex predicates, and improved support for `IN`.

It is important to note that with each version, the optimizer not only gained new capabilities — it became more predictable and manageable. New monitoring tools appeared, such as the `MON$COMPILED_STATEMENTS` table in Firebird 5.0, which allows peeking into the cache of compiled queries and seeing execution plans without the need to re-run the query.

What This Book Is About: A Roadmap by Chapters

This is not just a SQL reference or a description of Firebird's capabilities. This is a practical guide that systematically reveals the secrets of the optimizer and teaches you to think like the optimizer. Each chapter is built on the principle of "from theory to practice," where concepts are first explained, and then it is shown how to apply this knowledge to write efficient queries.

How to Measure Query Performance

The path to optimization begins with the ability to measure correctly. In the first chapter, you will learn to use Firebird's built-in tools for measuring performance. You will learn what the Reads, Writes, Fetches counters mean and how they relate to the actual query execution time. You will learn how to use `isql` to get statistics and understand the difference between a "cold" query start (when data is read from disk) and a "warm" one (when everything is already in the cache). This chapter lays the foundation for everything else because you cannot optimize what you cannot measure correctly. As Lord Kelvin said: "If you cannot measure it, you cannot improve it."

What is a Query Execution Plan

This is where the real immersion into the optimizer's work begins, the moment when the veil of mystery is lifted. You will learn what Legacy and Explain plans are, and most importantly — you will learn to read and interpret them, like an experienced detective reads clues. You will see how to get a query plan in various ways, including using `SET EXPLAIN` in `isql` and the monitoring tables `MON$STATEMENTS` and `MON$COMPILED_STATEMENTS`. Understanding the execution plan is a key skill, absolutely critical for success. Without it, all other knowledge about optimization will be incomplete, like trying to drive a car with your eyes blindfolded.

Access Methods Used in Firebird

This is one of the central and most important chapters of the book, the heart of the entire system of knowledge about the optimizer. Here, all methods that the optimizer can use to retrieve data are examined in detail, with examples and explanations: full table scan (Full Table Scan), various types of index access, bitmaps (Bitmap), range scan (Range Scan). For each method, the conditions for its use, advantages, disadvantages, and cost are explained in detail. You will understand why sometimes a full scan of a small table turns out to be faster than index access, and how the optimizer makes this decision based on statistics. Special attention is paid to the concept of index selectivity and how it critically influences the choice of access method.

This chapter was previously published as a white paper for the general public and contains key points about the optimizer's operation.

Query Transformations

The optimizer does not simply execute the query as you wrote it — it transforms the query into an equivalent but more efficient form. This chapter reveals the secrets of such transformations: how the optimizer works with the `LIKE` predicate, transforms subqueries, applies predicate inversion. Understanding these mechanisms allows you to write queries that are easier to optimize.

Table Statistics and gstat

Here you will learn about the critical importance of up-to-date statistics for the optimizer's correct operation. You will see how statistics are collected, what selectivity is, and how to use the `gstat` utility to analyze data distribution. You will learn to determine when statistics are outdated and require updating, and understand the connection between statistics and query execution plans.

Using Indexes

Indexes are a powerful tool, but only when applied correctly. This chapter systematizes all knowledge about indexes: when they are used, when they are ignored, how to create effective composite indexes, what partial indexes are and how they work. Separate attention is paid to conditions under which indexes are used for `ORDER BY` and `GROUP BY` without additional sorting.

Optimizing Joins (JOIN)

Table joins are often the most narrow and painful point in query performance, the source of a significant portion of problems. Here, all available join algorithms are examined in detail: the classic Nested Loop Join, the modern Hash Join, and Merge Join. You will learn how the optimizer determines the order of joining tables, what "join cost" is and how it is calculated, and most importantly — how you can influence the optimizer's decisions when it makes mistakes. Special attention is paid to the fundamental differences between `INNER JOIN` and `LEFT JOIN` from an optimization perspective — the difference here can be dramatic. You will understand when Hash Join will give you a 10x boost, and when it will remain unused.

Optimizing Sorts

Sorting can be a very expensive operation, especially for large volumes of data. In this chapter, you will learn when sorting is inevitable and when it can be avoided by using index navigation. This chapter analyzes the Refetch method, you will see how it helps reduce the volume of data to be sorted. You will understand the connection between memory for sorting (`TempCacheLimit`) and performance, and learn to optimize queries with `ORDER BY`.

Optimizing Groupings

Grouping with `GROUP BY` is another potentially expensive operation. Here, conditions for using indexes for grouping, ways to reduce grouping cost by moving it to a CTE, and optimization of aggregate functions are considered. You will see how the query structure affects grouping efficiency and learn to choose the optimal approach for each specific task.

Conclusion

The final part summarizes and discusses topics that remained outside the scope of this edition but will be covered in future versions of the book: optimizing procedural code, using the profiler, working with window functions, denormalization and stored aggregates.

How to Read This Book: Practical Recommendations

This book is written to be useful for readers with different levels of experience and various goals. Here are several usage scenarios:

For beginner developers:

If you are just starting with Firebird or have basic SQL knowledge, it is highly recommended to read the book sequentially, from the first chapter to the last, without skipping ahead. The material is carefully structured with increasing complexity: basic concepts and terminology are established first, then more complex and advanced topics are explained on this solid foundation. Do not rush and do not try to "skim through" the book over a weekend — take the time to deeply understand each concept, think through the examples, and experiment on your own test database.

It is especially important to thoroughly master the first three chapters: performance measurement, reading execution plans, and access methods. This is the foundation, the cornerstone on which the entire edifice of your optimization knowledge is built. If something remains unclear — do not hesitate to go back and reread it. Create your own test database with examples from the book, experiment by changing queries and carefully observing the changes in execution plans and statistics. Practice through experimentation is the best teacher.

For experienced developers:

If you already have experience with SQL and Firebird, you can use the book as a reference, consulting specific chapters as needed. Encountered a slow table join — read the chapter on JOIN. Problems with sorting — go to the corresponding section.

Nevertheless, we recommend at least skimming all chapters — you will likely find techniques and approaches you did not know about or had not considered. Pay special attention to the chapters on query transformations and the use of indexes — there are many non-obvious nuances there that can explain the optimizer's mysterious behavior in your queries.

For database administrators:

If your main task is ensuring the performance of existing systems, pay special attention to the chapters on statistics, gstat, and access methods. Understanding how the optimizer uses statistics to make decisions will help you configure databases correctly and create effective indexes.

The chapters on joins and sorting will also be useful — they will help identify bottlenecks in queries written by developers and suggest ways to optimize them.

For solving specific problems:

If you came to this book with a specific performance problem, here is a quick path:

1. Start with the chapter "How to Measure Performance" — make sure you are measuring the

problem correctly.

2. Study the chapter "What is an Execution Plan" — obtain the plan for your slow query.
3. Find the bottleneck in the plan (usually an operation with high cardinality or a high number of Reads).
4. Go to the corresponding chapter: if the problem is in a join — read about `JOIN`, if in sorting — about `ORDER BY`, if in a full scan of a large table — about access methods and indexes.

Practical exercises:

Although this book provides numerous examples, you will gain the most benefit if you experiment on your own. If you do not have a suitable-sized database at hand, create a test database, populate it with data (the more, the better — at least tens or hundreds of thousands of records in the main tables), and try to reproduce the situations described in the book.

It is best to take your real queries from work projects and analyze them using the knowledge gained. Obtain execution plans, look at the statistics, try to apply optimization techniques.

About examples and versions:

The examples in this book apply to Firebird versions 3.0 and above, with separate notes for features introduced in versions 4.0 and 5.0. If you are working with earlier versions, most of the material will still be relevant, but some features (for example, Hash Join) will not be available.

Tools for work:

The book uses standard Firebird tools: the command-line utility `isql`, tracing (`tracemgr`), the `gstat` utility. All of them are included with Firebird and do not require additional installation. Although you can use any graphical administration tool to execute queries and obtain statistics, the examples in the book are given for `isql` to eliminate dependency on third-party tools and show the work "in its pure form."

What You Will Get After Reading This Book

Upon completing the study of the material, you will possess systematic knowledge about how Firebird executes SQL queries. This is not just a collection of disparate techniques and tricks — it is a holistic, deep understanding of the optimizer's work, which will fundamentally change your approach to writing queries and designing databases.

You will learn to "think like the optimizer" — to anticipate which execution plan will be chosen for your query even before running it, and you will be able to write code in advance that will execute efficiently on the first attempt. You will stop relying on guesses, intuition, and trial and error, and will start making informed, confident decisions based on an understanding of the DBMS's internal mechanisms.

Practical skills you will gain and be able to apply every day:

- The ability to quickly identify query performance bottlenecks in minutes, not hours
- The ability to read and analyze query execution plans like an open book

- Understanding when, where, and what indexes to create for maximum efficiency
- The skill of optimizing complex multi-table joins using different JOIN algorithms
- The ability to effectively use sorting and grouping, avoiding expensive operations
- Knowledge of built-in tools for monitoring, diagnosing, and profiling queries
- The ability to predict optimizer behavior even before executing a query

But most importantly — you will gain confidence and professional freedom. Confidence that you can solve any performance problem, armed with knowledge and tools, rather than just blindly experimenting with different query variants in desperate hope for a random improvement. Confidence that the code you write will run fast not only on test data of ten records, but also in real production with millions of records, when user satisfaction and business success are at stake.

This book will not turn you into an optimization guru overnight — true mastery comes only with practice and experience. But it will give you an accurate map of the terrain, a reliable compass for navigation, and a proven set of professional tools for a successful journey into the world of high-performance SQL queries. The rest is in your hands.

Welcome to the fascinating world of the Firebird optimizer. There is no place for magic and chance here — only logic, understanding, and measurable results. Let’s begin this exciting journey to mastery!

Alexey Kovyazin, Vice President of the Firebird Foundation (2025-2026)

Chapter 1. How to Measure Query Performance?

The primary measure of query performance is its execution time. For SQL queries returning a cursor, its execution time is the time to fetch **all** the query's records.

In some cases, the responsiveness of the application is important, where the first batch of data is shown to the user as quickly as possible, and all cursor records might not be fetched at all. These are typically applications with data grids. In this case, the execution time can be measured as the time to get the first batch of data (the number of records may vary). Furthermore, such queries require a change in optimization strategy. Optimization strategies will be described in detail later in the corresponding section.

So how can you measure the query execution time? There are many tools for this. The simplest one is a timer in your application that gets the time before the query starts executing and after it completes, and then calculates the difference between these points. For example, the interactive tool `isql` calculates the execution time of SQL queries in this way. A more precise tool for measuring query performance is tracing, as it performs measurements on the server side.

Furthermore, there is a PSQL profiler, which also measures the execution time of queries and their parts. But it introduces significant overhead, as it can perform measurements tens of thousands of times per second, and is intended primarily for identifying bottlenecks in complex procedural logic. In this case, the accuracy of measuring the total query execution time is not as important as the ratio of the execution times of its individual parts (access methods or PSQL statements).

We will talk about tracing and the profiler later, but for now, let's learn how to measure query performance using `isql`.

1.1. Enabling Execution Statistics Output in isql

By default, `isql` only displays the query execution results, not the statistics. To enable the output of query execution statistics, you need to use the `SET STAT` command.

Syntax of the SET STAT command

```
SET STAT[s] [ON|OFF]
```

Let's see what is output when statistics are enabled.

```
SQL> connect inet://localhost/test user SYSDBA password 'masterkey';
Database: inet://localhost/test, User: SYSDBA
SQL> SET STAT ON;
SQL> SELECT COUNT(*) FROM HORSE;

                COUNT
=====================
```

```
               538832

Current memory = 551606960
Delta memory = 334112
Max memory = 551691824
Elapsed time = 3.491 sec
Buffers = 32768
Reads = 9468
Writes = 0
Fetches = 577191
```

As you can see, the query execution result is displayed first, followed by its execution statistics.

1.2. What Do These Numbers Mean?

- Current memory — this is the current memory consumption in bytes by the Firebird process.
- Delta memory — this is the change in memory consumption by the Firebird process after executing the current command. It can be negative if memory was returned to the OS.
- Max memory — this is the maximum memory consumption by the Firebird process.
- Elapsed time — the execution time of the last statement. It also includes the time for preparing the SQL statement, the execution time on the server side, and the time to return all cursor records.
- Buffers — the size of the page cache in pages. This value is constant at least within a session. Usually, it is equal to the value of the configuration parameter `DefaultDbCachePages` or the value specified in the database header.
- Reads — the number of pages read from disk.
- Writes — the number of pages written to disk.
- Fetches — the number of logical page reads. The number of reads from the page cache.

Let's try to execute the same query again.

```
SQL> SELECT COUNT(*) FROM HORSE;

                COUNT
=====================
               538832

Current memory = 551607056
Delta memory = 96
Max memory = 551691824
Elapsed time = 0.322 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 576500
```

As you can see, the execution statistics have changed. The first thing that catches the eye is the

different execution time. The first run was on a cold database with an unheated cache, so it was slow. The second run was fast. The page cache size was sufficient to hold all the pages, so the second run of the same query does not read any pages from disk at all. Everything is in the page cache.

Now let's disconnect and execute the same query again in a new connection.

```
SQL> connect inet://localhost/test user SYSDBA password 'masterkey';
Commit current transaction (y/n)?y
Committing.
Database: inet://localhost/test, User: SYSDBA
SQL> SELECT COUNT(*) FROM HORSE;

                COUNT
=====================
               538832

Current memory = 551590528
Delta memory = 334112
Max memory = 551675392
Elapsed time = 0.545 sec
Buffers = 32768
Reads = 9469
Writes = 0
Fetches = 577191
```

The statistics have changed again. The execution is clearly faster than the first time, but slower than the second. What's the matter?

As is known, besides its own page cache, Firebird can also use the operating system's file cache. This depends on the configuration parameters `UseFileSystemCache` or `FileSystemCacheThreshold`. In this case, we are using the SuperServer architecture, the connection to the database is the only one, and when the last connection to the database is broken, the page cache is freed. However, the OS file cache lives by different rules; the OS decides when it will be freed. In this case, part of the database still remained in the OS file cache, so despite the presence of `Reads`, the query executes quickly, although slower than when the necessary pages are in the page cache.

So which result should you use? Usually, more than one connection works with your system, and these connections are more or less permanent, so the most correct way is to compare query performance by the second run. Further, in the presentation of the materials in this book, we will always operate with the results of the second query run.

1.3. How to Get the Time to Fetch All Records

In the previous examples, we measured the performance of queries that returned only one record. In this case, the costs of displaying data in the console and transmitting data over the network were minimal, so the query execution time does not differ much from the obtained result. But what if you need to measure the execution time of a query returning hundreds of thousands of records?

First, let's try to measure and then minimize the costs of displaying data.

Let’s execute the following query, which will fill the page cache and display the total amount of data in the TRIAL table.

```
SQL> set stat on;
SQL> select count(*) from trial;

                COUNT
=====================
               160828

Current memory = 551532192
Delta memory = 254208
Max memory = 551616288
Elapsed time = 0.130 sec
Buffers = 32768
Reads = 1855
Writes = 0
Fetches = 168550
```

And now let’s execute a query that outputs one of the columns of this table.

```
SQL> select code_trial from trial;
...
           CODE_TRIAL
=====================
               486580
               486581
               486582
               486583
               486584
               486585
               486586
               486587

Current memory = 551555472
Delta memory = 23280
Max memory = 551628576
Elapsed time = 68.636 sec
Buffers = 32768
Reads = 1
Writes = 0
Fetches = 168098
```

Be patient, the query will take a long time to execute. A lot of data will be output in place of the ellipsis. This not only distorts the query execution time but also makes measurements with isql inconvenient. But there is a simple way out of this situation - you need to redirect the query results output to a null device. To do this, you need to execute the command.

```
OUTPUT nul;
```

Let's try again.

```
SQL> OUTPUT nul;
SQL> select code_trial from trial;
Current memory = 551555568
Delta memory = 96
Max memory = 551628576
Elapsed time = 0.903 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 168085
```

Thus, we have completely eliminated the delays for displaying data.

To return to the data display mode, type the command.

```
OUTPUT;
```

1.4. Query Transformation

How to eliminate network data transfer delays? Currently, there is no way.

However, when optimizing SQL queries, you can optimize not the original query but another transformed query and measure the execution time of this transformed query. After the optimization is complete, you can return to the original query.

Usually, such transformation boils down to

```
SELECT
  COUNT(*)
FROM (<source_query>)
```

where `<source_query>` is the original query for optimization.

Why is this particular form of transformation used? Because this query forces the retrieval of all records from the original query while returning exactly one record.

Let's say you are trying to optimize the following query

```
SELECT
  H.NAME
FROM
  HORSE H
  JOIN FARM ON FARM.CODE_FARM  = H.CODE_FARM
WHERE H.CODE_DEPARTURE = 1
  AND FARM.CODE_COUNTRY = 1
```

Let's perform the transformation mentioned above

```
SELECT
  COUNT(*)
FROM (
  SELECT
    H.NAME
  FROM
    HORSE H
    JOIN FARM ON FARM.CODE_FARM  = H.CODE_FARM
  WHERE H.CODE_DEPARTURE = 1
    AND FARM.CODE_COUNTRY = 1
)
```

The original query lacks parts that are computed after the `FROM` and `WHERE` clauses, so the query can be simplified.

```
SELECT
  COUNT(*)
FROM
  HORSE H
  JOIN FARM ON FARM.CODE_FARM  = H.CODE_FARM
WHERE H.CODE_DEPARTURE = 1
  AND FARM.CODE_COUNTRY = 1
```

Now you can optimize the query that returns exactly one record. You can create or delete indexes, use various hints, and compare the execution time of the new query. When it becomes better, switch back to the original query by replacing `COUNT(*)` with the list of original fields.

If the query is more complex, such simplifications cannot be applied. For example, we optimize the following query

```
SELECT
  MEASURE.CODE_HORSE,
  AVG(MEASURE.HEIGHT_HORSE) AS AVG_HEIGHT_HORSE
FROM
  MEASURE
WHERE EXTRACT(YEAR FROM MEASURE.BYDATE) = 2022
GROUP BY MEASURE.CODE_HORSE
```

then the contents of the `SELECT` clause cannot be replaced, so the transformed query will look like this

```
SELECT COUNT(*)
FROM (
  SELECT
    MEASURE.CODE_HORSE,
    AVG(MEASURE.HEIGHT_HORSE) AS AVG_HEIGHT_HORSE
  FROM
```

```
    MEASURE
  WHERE EXTRACT(YEAR FROM MEASURE.BYDATE) = 2022
  GROUP BY MEASURE.CODE_HORSE
)
```

When optimizing queries, such transformations will be used quite often, so I recommend remembering them.

1.5. Measuring Query Performance Under Load

What if you need to measure query performance under a working load, meaning when other connections are working with the database? This can be done in the same way as measuring a query without load, but there are nuances.

Let's see how load affects measurement results. First, let's measure the query performance without additional load.

```
SELECT COUNT(*) FROM HORSE;
```

We get the following result:

```
                COUNT
=====================
               525875

Current memory = 551664112
Delta memory = 96
Max memory = 551748880
Elapsed time = 0.117 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 562542
```

Now let's emulate a working load. To do this, start another isql session and run a resource-intensive SQL query in it:

```
OUTPUT NUL;
SELECT * FROM COVER, TRIAL_LINE;
```

A meaningless query performing a CROSS JOIN of large tables is intentionally written here, which guarantees that the query will run for a sufficiently long time.

Run the original query again:

```
SELECT COUNT(*) FROM HORSE;
```

And we get:

```
                COUNT
=====================
               525875

Current memory = 553062784
Delta memory = 0
Max memory = 553136800
Elapsed time = 0.126 sec
Buffers = 32768
Reads = 42
Writes = 0
Fetches = 566702
```

The execution time increased slightly, which is normal, but the Fetches counter also increased. But how can that be, the query is the same. Let's try running it again:

```
                COUNT
=====================
               525875

Current memory = 553062784
Delta memory = 0
Max memory = 553136800
Elapsed time = 0.128 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 566775
```

Again, a different value for Fetches. What's the matter? The matter is that Firebird operates in Super Server mode. In this mode, all I/O-related counters are shared. How are Reads, Writes, Fetches calculated for an executed query? The page cache has counters for these operations, which are always incremented. The query execution statistics show the difference between the values of these counters after and before the query execution. Thus, if your server is running in Super Server mode, the values of Reads, Writes, and Fetches under load will be random variables, and therefore no conclusions can be drawn based on them. Nevertheless, the query execution time under load can be measured.

In Firebird 5.0, this influence is much lower than in previous versions. The reason is the improvement CORE-7814.

In Classic and Super Classic modes, the page cache, and therefore its counters, are separate for each connection, so Reads, Writes, and Fetches will not differ for the case of a single connection and under a working load.

1.6. Using Tracing to Measure Query Performance

What to do if we still need to evaluate Reads, Writes, and Fetches under load in Super Server mode? For this, tracing can be used.

Unlike `isql` and other tools for obtaining query execution statistics, in tracing, the Reads, Writes, Fetches, and Marks counters are measured directly for the executed query, not taken from shared memory. Therefore, the values of these counters are more accurate and are not affected by other database connections.

Let's see how to measure query performance using tracing.

Before starting a trace session, you need to prepare a configuration file that will allow displaying query execution statistics. The configuration must be set up so that the trace only includes events from our database and only for the monitored connection. For this, use the following configuration file template:

```
database = <db_name>
{
    enabled = true
    log_initfini = false

    connection_id = <connection_id>

    log_statement_prepare = true
    log_statement_finish = true

    print_perf = true

    time_threshold = 0

    max_sql_length = 8000
    max_arg_length = 400
}
```

Where `<db_name>` is the path to the database or its alias. Here, instead of the full database name, you can write a regular expression. In this case, events from all databases whose file or alias name matches the regular expression will be included in the trace.

`<connection_id>` is the identifier of the monitored connection. It can be obtained with the following query:

```
SELECT CURRENT_CONNECTION FROM RDB$DATABASE;
```

```
   CURRENT_CONNECTION
======================
                   136
```

Now, you can create a configuration file for tracing:

```
database = test
{
    enabled = true
    log_initfini = false
    connection_id = 136
    log_statement_prepare = true
    log_statement_finish = true
    print_perf = true
    time_threshold = 0
    max_sql_length = 8000
    max_arg_length = 400
}
```

Let's save it under the name `test_trace.conf`.

Next, start an interactive trace session using the command:

```
fbtracemgr -SE inet://localhost/service_mgr -START -NAME test_trace -USER SYSDBA -PASS masterkey -CONFIG test_trace.conf
```

And now let's repeat our performance measurement experiment for the query:

```
SELECT COUNT(*) FROM HORSE;
```

The interactive trace session will display the following:

```
2024-09-18T20:05:13.1940 (3996:00000000032F17C0) PREPARE_STATEMENT
        test (ATT_136, SYSDBA:NONE, UTF8, TCPv6:::1/58645)
        c:\Firebird\5.0\isql.exe:6588
                (TRA_3300, READ_COMMITTED | READ_CONSISTENCY | WAIT | READ_WRITE)

Statement 160:
-------------------------------------------------------------------------------
SELECT COUNT(*) FROM HORSE
      0 ms

2024-09-18T20:05:13.3030 (3996:00000000032F17C0) EXECUTE_STATEMENT_FINISH
        test (ATT_136, SYSDBA:NONE, UTF8, TCPv6:::1/58645)
        c:\Firebird\5.0\isql.exe:6588
                (TRA_3299, CONCURRENCY | WAIT | READ_WRITE)

Statement 160:
-------------------------------------------------------------------------------
SELECT COUNT(*) FROM HORSE
1 records fetched
    108 ms, 562542 fetch(es)

Table                             Natural     Index    Update    Insert    Delete   Backout     Purge   Expunge
*******************************************************************************************************************
HORSE                              525875
```

Let's run the query emulating the working load in a new session:

```
OUTPUT NUL;
SELECT * FROM COVER, TRIAL_LINE;
```

In the session where we are conducting measurements, run the query again:

```
SELECT COUNT(*) FROM HORSE;
```

The interactive trace session will display the following:

```
2024-09-18T20:08:30.5300 (3996:00000000032F17C0) PREPARE_STATEMENT
        test (ATT_136, SYSDBA:NONE, UTF8, TCPv6:::1/58645)
        c:\Firebird\5.0\isql.exe:6588
                (TRA_3300, READ_COMMITTED | READ_CONSISTENCY | WAIT | READ_WRITE)

Statement 160:
-------------------------------------------------------------------------------
SELECT COUNT(*) FROM HORSE
      0 ms

2024-09-18T20:08:30.6450 (3996:00000000032F17C0) EXECUTE_STATEMENT_FINISH
        test (ATT_136, SYSDBA:NONE, UTF8, TCPv6:::1/58645)
        c:\Firebird\5.0\isql.exe:6588
                (TRA_3299, CONCURRENCY | WAIT | READ_WRITE)

Statement 160:
-------------------------------------------------------------------------------
SELECT COUNT(*) FROM HORSE
1 records fetched
    114 ms, 562542 fetch(es)

Table                             Natural     Index    Update    Insert    Delete   Backout     Purge   Expunge
*******************************************************************************************************************
HORSE                              525875
```

As you can see, the number of Fetches did not change. Additionally, per-table statistics for query execution are displayed. We will discuss what this is and how to analyze it later.

1.7. Conclusions

- When measuring query execution time, for queries returning a cursor always fetch all records.
- To reduce the impact of printing data to the `isql` console, redirect the output to a null device using the `OUTPUT nul` command.
- When optimizing queries, you can apply various transformations and measure the execution time of the transformed queries to test our hypotheses.
- If Firebird is running in SuperServer mode, then when measuring performance in `isql` or GUI tools, you can trust the Reads, Writes, Fetches, Marks counter values only if your connection is the only one.
- If you need accurate Reads, Writes, Fetches, Marks counter values under load, use custom

Parameter	Type	Description
RECORD_LENGTH	INTEGER	Record length in bytes.
KEY_LENGTH	INTEGER	Key length in bytes.
ACCESS_PATH	BLOB SUB_TYPE TEXT	Description of the access method used by the data source.

This implementation has the following advantages over a separate EXPLAIN statement:

- It does not require special support on the client side, meaning old applications and fbclient can use this feature immediately, without modification.
- The plan can be output in any convenient form, for example, it can be visualized graphically, as in some popular Oracle and MS SQL tools.

Furthermore, the procedure returns a new column — the data source cardinality. This information is currently not available when using other methods to get the plan.

Example of using the RDB$SQL.EXPLAIN procedure to get the plan:

Example 1. Using the RDB$SQL.EXPLAIN *procedure*

```
SELECT *
FROM RDB$SQL.EXPLAIN(Q'{
  SELECT *
  FROM
    HORSE
    JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
    JOIN BREED ON BREED.CODE_BREED = HORSE.CODE_BREED
  WHERE HORSE.CODE_DEPARTURE = ?
}')
```

Currently, Firebird 6.0 is under active development. All its features have not yet taken their final form and may change by the release. You can “play around” with the plan output using the RDB$SQL.EXPLAIN procedure. I will use some of its capabilities, in particular the information about the cardinality of data sources when explaining access methods.

2.6. Conclusion

In this chapter, you learned about all the known ways to get query execution plans, as well as cursor plans inside stored procedures. We also looked into the near future and saw some developments in Firebird 6.0 that simplify getting the query plan and expand information about access methods.

Chapter 3. Data Access Methods Used in Firebird

Now that you know how to obtain an SQL query plan, let's learn how to read it. For this, it is necessary to become familiar with the data access methods used by the Firebird core.

This chapter describes the data access methods existing in **Firebird 5.0**. It is based on the original article Firebird: Data Access Methods by Dmitry Yemanov.

Compared to the original article, the following has been changed:

- Added descriptions of data access methods that appeared after Firebird 2.0 — in 2.1, 2.5, 3.0, 4.0, and Firebird 5.0.
- The schematic query execution tree has been replaced with real Explain plans (appeared in Firebird 3.0).
- Since the original article provided examples of calculating cardinality and cost, Explain plans have been supplemented with them. 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 obtained by me when analyzing the Firebird source code.
- A formula for calculating the cost of Hash Join has been added.
- Added a description of how filters reduce cardinality (Filter, group by, distinct).

3.1. Terminology

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

From the point of view of the types of operations performed, there are three classes of data sources:

- primary data access method — performs reading from a table or stored procedure, forms the initial data stream;
- filter — transforms one input data stream into one output data stream;
- merge — converts two or more input data streams into one output data stream.

Data sources can be pipelined and buffered. A pipelined data source outputs records during the reading of its input streams, while a buffered source must first read all records from its input streams and only then can output the first record.

From a performance evaluation perspective, each data 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 data access method. The cost value directly

depends on the cardinality and the mechanism of selecting or transforming the data stream. In current server versions, the cost is determined by the number of logical reads (page fetches) required to return all records by the data access method. Thus, "higher-level" methods always have a greater cost than low-level ones. CPU load in computational data access methods is converted to an approximately equivalent (according to given coefficients) number of logical reads.

3.2. Primary Data Access Methods

This group of data access methods performs the creation of a data stream based on low-level sources, such as tables (internal and external) and procedures. Next, we will consider each of the primary data sources separately.

3.2.1. Reading a Table

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

Full Table Scan

This method is also known as Natural Scan or Sequential Scan.

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

Since the expected data volume is quite large, there is a problem during the reading of table pages from the disk: the read pages can displace other pages, potentially needed by competing sessions. To handle this, the logic of the page cache changes — the current page of the scan is located at 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 page 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.

Records are read from the table one by one, immediately being output. It should be noted that there is no pre-fetching of records (even within a page) with their buffering in the server. That is, a full selection from a table with 100K records occupying 1000 pages will result in 100K page fetches, not 1000, as one might assume. There is also no multi-block reads, where adjacent pages could be grouped and read from the disk in “batches” of several at a time, thereby reducing the number of physical I/O operations.

Both of these capabilities are planned for implementation in future versions of the server.

Starting from Firebird 3.0, Data Pages (DP) for tables (containing more than 8 DP) are allocated in groups called **extents**. One extent consists of 8 pages. This allows

> the engine to use OS prefetch more efficiently, since pages belonging to the same 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.

When choosing this data access method, the optimizer uses a rule-based strategy — a full scan is performed only in the absence of indexes applicable to the query predicate. The cost of this data access method is equal to the number of records in the table (estimated approximately by the number of table pages, record size, and average record compression ratio on data pages). Theoretically, this is incorrect and the choice of data 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 later, when mentioning optimizer behavior, the following will be provided: an example SELECT query, its execution plan in Legacy and Explain forms. In the Legacy execution plan, a full table scan is denoted by the word “NATURAL”. In the Explain form — `Table "<table name>" Full Scan`. Currently, the values in square brackets (cardinality and cost) are not output in the explain plan; they are provided as a calculation example.

Example 2. Full Scan of the 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 query execution statistics, records read by a full scan are reflected as non-indexed reads.

When using table aliases, the plan will also display the aliases.

Example 3. Full Scan of the RDB$RELATIONS Table with an 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
```

Access by Record Identifier

In this case, the execution subsystem obtains the identifier (physical number) of the record that needs to be read. This record number can be obtained from various sources, each of which will be described later.

Somewhat simplified, the physical record number contains information about the page on which the record is located and the offset within that page. Thus, this information is 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 for bitmap-based retrieval (both index scans and access via `RDB$DB_KEY`) and index navigation. These access methods will be described in more detail later.

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

Positioned Access

This refers to positioned `UPDATE` and `DELETE` commands (the `WHERE CURRENT OF` syntax). There is a common misconception that this access method is a syntactic equivalent of retrieval using `RDB$DB_KEY`, but this is not the case. Positioned access works only for an active cursor, i.e., for a record that has already been fetched (using `FOR SELECT` or `FETCH` commands). Otherwise, an `isc_no_cur_rec` error ("no current record for fetch operation") will be raised. Thus, it is simply a way to reference the active record of a cursor, requiring no read operations at all. Whereas retrieval via `RDB$DB_KEY` utilizes access by record identifier and, consequently, always leads to the fetching of one page.

3.2.2. Index Access

The idea of index access is simple – besides the data table, we have another structure containing "key - record number" pairs in a form that allows for fast search by key value. In Firebird, an index is a paged B+ tree with prefix compression of keys.

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

- it can be used for predicates (`A = 0`) or (`A = 0 and B = 0`) or (`A = 0 and B = 0 and C = 0`), but 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 lead to a partial match on two segments, and the predicate (`A > 0 and B = 0`) – to 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 in some cases can limit themselves to only index pages – for example, if all selected fields are included in the index. But such a scheme is impossible in Firebird due to its architecture – the index key contains only the record number without information about its version – consequently, the server must always read the record itself to determine if at least one of its versions 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 it would be possible to implement a pure index-only scan. But there are two problematic points here. First, the key length would increase, hence the index would occupy more pages, leading to more I/O for the same scan operation. Second, every change to a record would lead to a modification of the index key, even if non-key fields were changed. Whereas currently, the index is modified only when the key fields of the record are changed. The first problem would lead to a significant performance degradation for scanning indexes with short keys (with types like INT, BIGINT, etc.). The second – at least triples the number of modified pages for every data modification command compared to the current situation. So far, the server developers consider this too high a price to pay for implementing a pure index-only scan.

There is an alternative variant of index-only scanning implemented in Postgres. It works effectively for tables where only a small portion of the records are changed. For this, besides 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 a bit in the visibility map for the corresponding data page. If it is set, it means this row is visible, and the data can be returned immediately. Otherwise, it will have to visit the row record and check if it is visible, so there will be no gain compared to a regular index scan.

This variant could possibly be implemented in Firebird as well. Starting from Firebird 3.0, for DP (Data Page) pages, there is a so-called swept flag. It is set to 1 if the sweep or garbage collector has visited the page and found no or cleaned up all non-primary record versions. This same flag is also present on PP (Pointer Page) pages, which could be used as an analog of the aforementioned visibility map.

Compared to other DBMSs, indexes in Firebird have another peculiarity – index scanning is always unidirectional, from smaller keys to larger ones. Often because of this, the index is called unidirectional, and it is said that its nodes have pointers only to the next node and no pointer to the previous one. Actually, the problem is not in that. All disk structures of the Firebird server are designed to be deadlock-free, and the minimum possible granularity of locks is guaranteed. In addition, the rule of "careful write" of pages is also in effect, serving for instant recovery after a crash. The problem with bidirectional indexes is that they violate this rule when splitting a page. At the moment, there is no known way for lock-free work with bidirectional pointers in the case where one page must be written strictly before another.

Actually, workarounds for this problem exist and are currently being discussed by the developers.

This peculiarity leads to the impossibility of using an ASC index for DESC sorting or calculating `MAX` and, conversely, the impossibility of using a DESC index for ASC sorting or calculating `MIN`. Of course, unidirectionality does not hinder index scanning for search purposes.

Index Selectivity

The main parameter affecting the optimization of index access is **selectivity**. It is a value inversely proportional to the number of unique key values. Cardinality and cost calculations assume a uniform distribution of key values in the index.

The index selectivity can be found using the following query

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

Example 4. Retrieving stored statistics for the CUSTNAMEX index

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

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

For composite indexes, besides the index selectivity, Firebird also stores the selectivity of the combination of its segments, starting from the first up to the given one. The selectivity of the segment combination can be found by querying the RDB$INDEX_SEGMENTS table.

Example 5. Retrieving stored 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 6. Retrieving stored statistics for the NAMEX index

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

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

Index selectivity is calculated when it is built (`CREATE INDEX`, `ALTER INDEX ⋯ ACTIVE`) and can become outdated over time. To update index statistics, the following SQL query is used

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

Example 7. Collecting statistics for the NAMEX index

```
SET STATISTICS INDEX NAMEX;
```

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

Stored statistics will be expanded in future versions of Firebird. Additionally, some capabilities for its automatic updating are planned.

In addition to regular and composite indexes, Firebird allows the creation of expression indexes. In this case, instead of table field values, the value of an expression returning a scalar value is used as the index keys. For the optimizer to use such indexes, the query's filter condition must use the exact same expression that was specified when creating the index. Expression indexes cannot be composite, but the expression can contain multiple table fields. The selectivity of such an index is calculated in the same way as for regular indexes.

Partial Indexes

Starting with Firebird 5.0, when creating an index, it is possible to specify an optional WHERE clause that defines a search condition limiting 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 the exact same logical expression as defined for the index;
- the search condition defined for the index contains logical expressions combined with 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`.

Let me demonstrate the last point with an example. Suppose you 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 predicates `=`, `<`, `>`, `BETWEEN`, and others, 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 will be used
SELECT *
FROM HORSE
WHERE DEATHDATE IS NOT NULL

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

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

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

If both a regular index and a partial index exist for the same set of fields, the optimizer will in most cases choose the regular index, even if the `WHERE` condition includes the same expression defined in the partial index. The reason for this behavior is that the selectivity of the regular index is known “precisely” (calculated as the inverse of the number of unique keys). However, the partial index contains fewer keys due to the additional filter condition, so its stored selectivity will be worse. The estimated selectivity of the partial index is evaluated as the stored selectivity multiplied by the fraction of records that made it into the index. This fraction is currently not stored in the statistics (“precisely” unknown) and is estimated based on the selectivity of the filter expression specified when creating the index (see Predicate Checking). At the same time, the actual selectivity may be lower, and therefore the regular index (with a precisely calculated selectivity value) may win.

This may change in future server versions.

Bitmaps

The main standard problem of index access is random input/output relative to data pages. Indeed, the order of keys in the index very rarely matches the order of the corresponding records in the table. Thus, fetching a significant number of records through an index will almost certainly lead to multiple fetches of each corresponding data page. This problem is solved in other DBMSs using clustered indexes (MSSQL term) or index-organized tables (Oracle term), where the data is located

directly in the index in ascending order of the cluster key.

In Firebird, this problem is solved differently. Index scanning is not a pipelined operation but is performed for the entire search range, including the obtained record numbers into a special bitmap. This map is a sparse bit array where each bit corresponds to a specific record, and the presence of a one in it indicates that this record should be fetched. The peculiarity of this solution is that the bitmap is by definition sorted by record numbers. 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 occurs in the physical order of the pages, as in a full scan, meaning each page will be read no more than once. Thus, the simplicity of index implementation here combines with maximally efficient access to records. The cost is a slight increase in response time for queries with a FIRST ROWS strategy, where the first records need to be obtained very quickly.

Operations of intersection (bit and) and union (bit or) are allowed on bitmaps, thus the server can use multiple indexes for one table.

Range Scan

Search by index is performed using upper and lower bounds. That is, if a lower scan bound is specified, the corresponding key is first found, and only then does the sequential enumeration of keys begin, with record numbers being added to the bitmap. If an upper bound is specified, each key is compared with it, and upon reaching it, the scanning stops. This mechanism is called a range scan. When the lower bound key equals the upper bound key, it is called an equality scan. If an equality scan is performed on a unique index, it is called a unique scan. This type of scan is of particular importance, as it can return no more than one record and, by definition, is the cheapest. If neither bound is specified, we are dealing with a full scan. This method is used exclusively for the index navigation described below.

When possible, the server skips NULL values during range scanning, if permissible. In most cases, this increases performance due to a smaller bitmap size and, accordingly, fewer indexed reads. This option is not used only for indexed predicates of the type `IS NULL` and `IS NOT DISTINCT FROM`, which consider NULL values.

When selecting indexes for scanning, the optimizer uses a cost-based strategy. The cost of a range scan is estimated based on 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 set in the code as a constant equal to 3.

In the Legacy execution plan, index range scanning is denoted by the word “INDEX”, followed in parentheses by the names of all indexes forming the final bitmap.

In the Explain plan, the display of range scanning 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 used, `scan_type` is the scan type, `table_name` is the table name.

Unique scan

Unique scan can only be used for unique indexes when scanning across all segments (full match) and using only equality predicates "=". Unique indexes are automatically created for primary key constraints or uniqueness constraints, but can also be created independently. This type of scan is particularly significant because it can return no more than one record, and therefore its cardinality is always 1. The cost of this scan is calculated by the formula

```
cost = indexDepth + 1
```

Example 8. 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
```

Equality Scan

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

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

```
cardinality = table_cardinality * index_selectivity
```

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

The cost of an index equality scan is calculated a bit more complexly:

```
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 size of the index page header, currently 39 bytes.

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

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

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

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

Example 9. 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 of a full match for a composite index:

Example 10. 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 two segments is used for a composite index.

Example 11. 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, then the Explain plan will show which scan bounds are used: "lower bound" for the lower bound, "upper bound" for the upper bound. For each bound, the number of index segments used is indicated. For example, (lower bound: 1/2) indicates that one segment out of two will be used for the lower bound.

The cost and cardinality of a range scan are calculated exactly the same as for an equality scan, but instead of index selectivity, the predicate selectivity is used, which is calculated as follows.

Table 5. Selectivity of index scan predicates

Predicate	factor	Selectivity
<, <=, >, >=	0.05	indexSelectivity + (1 - indexSelectivity) * factor
BETWEEN	0.0025	indexSelectivity + (1 - indexSelectivity) * factor
STARTING WITH	0.01	indexSelectivity + (1 - indexSelectivity) * factor
IN		indexSelectivity * N

Predicate	factor	Selectivity
Others	0.01	indexSelectivity + (1 - indexSelectivity) * factor

Example 12. 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 13. 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 14. Index Range Scan with lower and upper bounds

```
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 the bounds.

Example 15. 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 16. Range Scan of a composite index. Lower bound uses both segments, 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 bound of the first segment is used, and the second segment is not used at all.

Now let's try to use 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, it was not possible to use an index scan.

In some DBMSs (in particular, Oracle), there is a so-called Index Skip Scan, where the entire key of a composite index is not compared, but only the used part of the key is compared; in this case, the last example could have used an index scan. Currently, this access method is not available in Firebird.

List scan

List scan is available starting from Firebird 5.0. It is applicable only for the `IN` predicate with a list of values. In this case, a single shared bitmap is formed for the entire list of values. The search can be performed by vertical scanning (from the root for each key) or by horizontal scanning of the range between min/max keys. The optimizer decides exactly how the scanning will be performed based on which 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. Constant lists in `IN` are preliminarily 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 for the predicate, then list scan 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 list scan is calculated as follows:

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

Example 17. 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)
```

Bitmap intersection and union

As mentioned above, bitmap intersection (bit and) and union (bit or) operations are allowed, thus the server can use multiple indexes for a single table. When bitmaps are intersected, the resulting cardinality does not increase. For example, for the expression `F = 0 and ? = 0` its second part is not indexable and, consequently, is checked after the index selection, not affecting the final result. But bitmap union leads to an increase in the resulting cardinality, so only parts of a fully indexed predicate can be unioned. That is, if the second part of the expression `F = 0 or ? = 0` is moved to a higher level, it might turn out that all records needed to be scanned. Therefore, the index for field F will not be used in such an expression.

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

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

The selectivity of the union of two bitmaps is calculated by the formula:

```
bestSel + worstSel
```

Since the cost of access via the record identifier is one, the total cost of access via the bitmap will be equal to the total cost of the index search (for all indexes forming the bitmap) plus the resulting cardinality of the bitmap.

Let's look at examples using multiple indexes.

Example 18. Bitmap intersection

```
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 19. Bitmap union

```
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)
```

Index navigation

Index navigation is nothing more than a sequential scan of index keys. And since for each record the server needs to perform a record fetch and check its visibility for our transaction, this operation turns out to be quite expensive. This is precisely why this access method is not used for ordinary selections (unlike Oracle, for example), but is used only in cases where it is justified.

Sorting using multiple indexes is not supported.

Currently, there are two such cases. The first is the calculation of the aggregate functions `MIN`/`MAX`.

Obviously, to calculate `MIN` it is enough to take the first key in an ASC index, and to calculate `MAX` - the first key in a DESC index. If after fetching the 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 traverse the index, selecting records as we scan. In both cases, record fetching is performed based on access via its identifier.

About the unidirectionality of index navigation is written in Index Access.

There are several features that optimize this process. Firstly, if there are restricting predicates on the sort field, they create an upper and/or lower scan boundary. 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 reduces the costs of the scan itself. Secondly, if there are other restricting predicates (not on the sort field), but optimized via an index, a complex mode of operation is activated, where index scanning is combined with the use of a bitmap. Let's consider how this works. Suppose we have a query of the form (`WHERE A > 0 AND B > 0 ORDER BY A`). In this case, first, a range scan for the index on field `B` is performed and a bitmap is compiled. Then the value `0` is set as the lower boundary for scanning the index on field `A`. Then we scan this index starting from the lower boundary and for each record number extracted from the index, we check its inclusion in the bitmap. And only in case of inclusion do we perform the record fetch. This reduces the costs of fetching data pages.

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

Estimating the cost of navigation is quite difficult. For calculating `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 a negligibly small value, since in practice the described calculation of `MIN/MAX` will always be faster than alternative options. To estimate the cost of index navigation, it is necessary to take into account both the number and average width of the index keys, and the cardinality of the bitmap (if one exists), as well as have an idea of the index's clustering factor - the coefficient of correspondence between the location of keys and the physical numbers of records. Currently, the server lacks navigation cost calculation, meaning a rule-based strategy is used again. In the future, it is planned to use this approach only for `MIN/MAX` and `FIRST`, and in other cases rely on cost.

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

In Firebird 6.0, cost-based estimation was added for choosing between index navigation and external sorting.

In the Legacy plan, index navigation is denoted by the word "ORDER", followed by the name of the index (without parentheses, as there can be only one such index). The server also reports the indexes forming the bitmap used for filtering. In this case, both words are present in the execution plan: first "ORDER", then "INDEX".

Example 20. 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 21. Index navigation with a predicate creating a lower scan boundary

```
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 22. 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)
```

3.2.3. Access via RDB$DB_KEY

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

From the perspective of access methods, everything is simple here — a bitmap is created and a single record number — the value of RDB$DB_KEY — is placed into it. After that, the record is read using the access via record identifier described above. Obviously, the cost of such access is one. That is, we get something like pseudo-indexed 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 reflected in statistics as indexed. The reason for this is clear from the description above — everything read via access by record identifier is considered by the server as indexed access. A not entirely accurate, but fairly harmless assumption.

3.2.4. External table (External table scan)

This type of access is used only when working with external tables. Essentially, it is an analog of a

full table scan. Records are read one by one from an external file via the current pointer (offset). Page cache is not used. Indexing of external tables is not supported.

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

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

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

Example 23. 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
```

3.2.5. Virtual table (Virtual table scan)

This access method is used for reading data from virtual tables (monitoring tables MON$, security virtual tables SEC$, as well as the virtual table `RDB$CONFIG`). Data for such tables is generated on the fly and cached in memory. For example, all monitoring tables are populated upon the first access to any of them, and their data is retained until the end of the transaction. The virtual table `RDB$CONFIG` is also populated upon first access, and its data is retained until the end of the session.

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

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

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

Example 24. Reading the virtual table MON$ATTACHMENT

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

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

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

3.2.6. Local temporary table (Local table)

This access method is available starting from 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 statement INSERT ⋯ VALUES .. RETURNING, which can only insert and return one record, does not belong 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 row selection from the DML statement is finished.

In future versions of Firebird, it is planned 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 for accessing local tables, cost is not calculated and not used. The cardinality of local temporary tables is assumed to be 1000.

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

Example 25. 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
```

3.2.7. Procedural access

This access method is used when selecting from stored procedures that use the SUSPEND clause to return a result. In Firebird, a stored procedure is always a black box, about whose internals and implementation details the server makes no assumptions. A procedure is always considered a non-deterministic data source, meaning it can return different data in two subsequent calls executed under equal conditions. In light of this, it is obvious that any kind of indexing of the procedure's results is impossible. The procedural access method also represents an analog of a full table scan. With each fetch from the procedure, it is executed from the point of the previous suspension (the beginning of the procedure for the first fetch) until the next SUSPEND clause, after which its output parameters form a data row, which is returned from this access method.

Similar to access for external tables, the cost of procedural access is also not calculated and not taken into account, as there are no alternative options. For stored procedures, the cardinality is assumed to be 1000.

In the Legacy execution plan, procedural access is displayed as "NATURAL".

In older versions of Firebird (before 3.0), detailing (plans) of selections inside the procedure might be displayed. This allows evaluating the query execution plan as a whole, taking into account the "internals" of all involved procedures, but formally this is incorrect.

Example 26. Procedural access

```
SELECT *
FROM PROC_TABLE
```

```
PLAN (PROC_TABLE NATURAL)
```

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

3.3. Filters

This group of access methods acts as a transformer of the received data. The main difference of this group from 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 defined by their function. The input to a filter can be a data stream from a primary or any other data source.

From an implementation perspective, filters have another common trait — no cost is estimated for them. That is, it is considered that all filters execute in zero time.

Next, the existing implementations of filter methods in Firebird and the tasks they solve will be described.

3.3.1. Predicate Checking

This is probably the most frequently used case, and also the simplest to understand. This filter checks a certain logical condition for each record passed to its input. If this condition is met, the record is passed to the output unchanged; otherwise, it is ignored. Depending on the input data and the logical condition, fetching one record from this filter may result in one or more fetches from the input stream. Degenerate cases of checking filters are predefined conditions like (1 = 1) or (1 <> 1), which either simply pass the input data through themselves or remove it.

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

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

Table 6. Predicate Selectivity

Predicates	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 `AND` operator 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 output.

Example 27. Predicate filter check

```
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 is also displayed when using indexed access.

Example 28. Predicate filter check 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
```

Regarding the last example, a question may arise: why is a Filter used if the predicate is implemented via `INDEX UNIQUE SCAN`. The point is that Firebird implements fuzzy index search. That is, index scanning is merely an optimization for predicate calculation and in some cases may return more records than required. This is precisely why the server does not rely solely on the result of the index scan and checks the predicate again after fetching the record. Note that in the vast majority of cases, this does not incur noticeable performance overhead. In this case, the predicate check does not change the cardinality estimate, as it was already obtained during index access (filtered by the index).

Invariant Predicate Checking

A predicate is invariant if its value does not depend on the fields of the filtered streams. The simplest examples 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 also deterministic predicates are recognized by the optimizer and calculated once. Unlike regular filtering predicates, invariant predicates are calculated as early as possible; their check is always placed as “high” as possible in the execution tree. If an invariant filtering predicate has the value `FALSE`, then fetching records from the underlying data sources is immediately stopped.

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

In the Legacy plan, invariant predicate checking is not displayed. In the Explain plan, invariant predicate checking is displayed using the word “Filter (preliminary)”, but the predicate itself is not output.

Example 29. Invariant predicate checking

```
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
```

3.3.2. Sorting

Also known as external sort. This filter is applied by the optimizer when it is necessary to order the input stream if index navigation cannot be applied (see Index navigation). Examples of its use: sorting or grouping (when there are no suitable indexes or indexes are not applicable for the input stream), building a B+ tree index, performing the DISTINCT operation, preparing data for a single-pass merge (see below), and others.

Since the input data is by definition unordered, it is obvious that the sort filter must fetch all records from its input stream before it can output even one of them. Thus, this filter can be

considered a buffering data source.

External sorting is performed as follows. The set of input records is placed into an internal buffer, after which it is sorted by the quick sort algorithm and the block is moved to external memory. Next, the next block is filled in the same way and the process continues until the records in the input stream are exhausted. After that, the filled blocks are read and a binary merge tree is built from them. When reading from the sort filter, the tree is parsed and the records are merged in a single 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 (i.e., those specified in the `ORDER BY` or `GROUP BY` clause) and non-key fields (all other fields referenced within the query) into sort blocks, which are either kept 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 “truncating”. The first one preserves records with duplicate sort keys, while the second causes duplicates to be removed. It is the “truncating” mode that implements the DISTINCT operation, for example.

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

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

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

Memory consumption for sorting can be estimated by multiplying the record length by the cardinality of the sorted stream. By default, Firebird will attempt to perform the sort in RAM; as soon as this memory is exhausted, the engine will start using temporary files. The amount of RAM available for sorts is set by the `TempCacheLimit` parameter. In the Classic architecture, this limit is set per connection, while in the SuperServer and SuperClassic architectures, it is set for all connections to the database.

Example 30. Using external sort 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 31. Using external sort in truncating 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 32. Using external sort combined 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 cases of redundant sorting (for example, `DISTINCT` and `ORDER BY` on the same field) are eliminated by the optimizer and result in only the minimally necessary number of sorts.

Refetch

Starting with Firebird 4.0, a sorting optimization for wide data sets was introduced, which appears in the Explain plan as `Refetch`. A “wide data set” refers to a data set where the total length of the record fields is large.

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

The alternative approach (`Refetch`) is that only the key fields and the `DBKEY` of the records of all participating tables are stored inside the sort blocks, and the non-key fields are fetched from the data pages after sorting. This improves sort performance in the case of long non-key fields. From the description of `Refetch`, it is clear that it can only be applied in the case of a normal sort mode; `Refetch` is not applicable for the truncating sort mode because in this mode `DBKEY`s do not get into the sort blocks.

For external sorting using `Refetch`, the cost is not calculated. To choose the method for sorting data, the optimizer uses the `InlineSortThreshold` parameter. The value specified for `InlineSortThreshold` determines the maximum size of the sort record (in bytes) that can be stored inline, i.e., inside the sort block. Zero means that records are always refetched (Refetch). The optimal value for this parameter must 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 node is the word “Refetch”, and the lower level contains the word “Sort”. Furthermore, in parentheses, the length of the sorted keys + `DBKEY`s of the used tables (record length) and the sort key length (key length) are specified.

Example 33. Optimization of 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 the sort keys occurs, the sort keys themselves + the table's DBKEY are placed into the sort blocks, and then the full records are fetched from the table in sorted order by DBKEY using the `Refetch` method.

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

Example 34. In truncating 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, despite the need to sort wide records, `Refetch` cannot be used, so a regular external sort is applied.

3.3.3. 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 accumulate this register. In the case of grouping, the algorithm becomes somewhat more complicated. For the correct calculation of 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 it changes, 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 an external sort or index navigation (if possible). The Refetch method cannot be used to order the input stream for grouping.

There is an alternative method for calculating aggregates — hashing. In this case, sorting the input stream is not required; instead, a hash function is applied to each grouping key and a register is stored for each key (or rather, for each key collision) in a hash table. The advantage of this

algorithm is that no additional sorting is needed. The disadvantage is the greater memory consumption for storing the hash table. This method usually outperforms sorting when the selectivity of the grouping keys is low (few unique values, hence a small hash table). Hash aggregation is absent in Firebird.

Hash aggregation is planned for implementation in future server versions.

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

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

In the Legacy execution plan, aggregation is not shown.

In the Explain plan, aggregation is displayed using the word “Aggregate”.

Example 35. Calculation of 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 an ASCENDING index exists for the `RDB$RELATION_ID` field, the `MIN` aggregate function can be calculated using index navigation (see Index navigation).

Example 36. Calculation of 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 37. Calculation of an aggregate function with grouping. External sorting is used to separate 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 38. Calculation of 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 39. Calculation of 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 SUM/AVG/COUNT functions in DISTINCT mode, the values are sorted in duplicate elimination mode for each group before accumulation. In this case, the word SORT is not displayed in the plan.

Example 40. Calculation of 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
```

Filtering in the HAVING Clause

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

aggregate calculation. If the predicate is applied to grouping fields and expressions, the predicate will be pushed down, meaning it will be evaluated before grouping and aggregation.

Example 41. Filtering by an aggregate function in the 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 42. Filtering by a 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 aggregate calculation,

filtering was applied again. It should be noted that the repeated filtering performed cardinality normalization (in theory, aggregate functions with and without grouping can never return less than one record).

3.3.4. Counters

This type of filter is also very simple. Its goal is to output only a part of the records from the input stream, based on some value N of an internal counter. There are two varieties of this filter, used to implement the FIRST/SKIP/ROWS/FETCH FIRST/OFFSET clauses of a query. The first variety (FIRST counter) outputs only the first N records from its input stream, after which it outputs an EOF indicator. The second variety (SKIP counter) ignores the first N records from its input stream and starts outputting them beginning from record N+1. Obviously, if the query has a selection limit of the form (`FIRST 100 SKIP 100`), the SKIP counter must be applied first, and only after it the FIRST counter. The optimizer guarantees the correct application of counters during query execution.

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

In the Legacy execution plan, counters are not displayed.

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

Example 43. 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 44. 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 45. Using the 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 46. 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
```

3.3.5. Singularity Check

This filter is used to guarantee that only one record is returned from the data source. It is applied when singleton 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 the first record is output. Otherwise, it raises the `isc_sing_select_err` error ("multiple rows in singleton select").

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

In the Legacy execution plan, the singularity check is not displayed.

In the Explain plan, the singularity check is displayed as "Singularity Check".

Example 47. Singularity check 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 the Legacy form, the first plan refers to the subquery, and the second to the main query.

Example 48. Singularity check of a non-correlated 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)
```

3.3.6. Record Locking

This filter implements pessimistic record locking. It is applied only when the `WITH LOCK` clause is present in the query text. Each record read from the filter's input stream is returned to the output already locked by the current transaction. For locking, a new version of the record is created, marked with the identifier of the current transaction. If the current transaction has already modified this record, locking is not performed.

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

Example 49. Record locking

```
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
```

3.3.7. Conditional Stream Branching

Conditional stream branching has been available since Firebird 3.0. This access method is used in cases where, depending on an input parameter, a table can be read either via a full scan or using index access. This is usually possible for a filter condition of the type `INDEXED_FIELD = ? OR 1=?`.

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

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

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

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

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 50. Conditional Stream Branching

```
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
```

3.3.8. Record Buffering

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

Record buffering is intended to store the results of the underlying access methods in RAM; as soon as 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 per connection, while in the SuperClassic and SuperServer architectures, it is set for all connections to the database.

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 displayed as “Record Buffer”, followed by the record length in parentheses as (`record length: <length>`).

Examples that use record buffering will be shown when describing the “Sliding Window” and “Hash Join” access methods.

3.3.9. Sliding Window (Window)

This group of access methods is used when computing so-called window or analytical functions.

A window function performs calculations for a set of rows that are in some way 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, as 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 not only the current row of the query result. The set of rows over which the window function performs calculations is called a **window**.

The set of rows can be divided into partitions. By default, all rows in the window are included in one partition; this can be changed by specifying in `PARTITION BY` how the rows are divided into partitions. For each row, the window function only processes the rows that fall into the same partition as the current row.

Within each partition, the order in which rows are processed by the window function can be specified. This can be done in the window expression using `ORDER BY`.

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

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

From the above, it follows that the access methods used when computing window functions do not change the cardinality of the result set. The cost of access methods for computing window functions is not calculated because there are no alternative options.

The computation of window functions begins with buffering the result set, which is performed

after all other parts of the SELECT query, but before sorting (ORDER BY), limiting the result set using first/skip counters, and computing expressions for columns in the SELECT clause. This buffer is called the window buffer. In the Explain plan, it is denoted as Window Buffer (see also Record Buffering).

Next, the boundaries of the partitions are defined in the window buffer, which is denoted in the Explain plan as Window Partition. This is done even if the PARTITION BY clause is absent (there will be one partition).

If the window expression describes partitioning using PARTITION BY, then the original (one partition) will be split into partitions again. For partitioning, external sorting by the partitioning keys is used. External sorting is also used to define the row processing order (ORDER BY within the window). Firebird combines the partitioning keys and the keys for defining the row order into one external sort. If the row set for external sorting is too wide, the Refetch method can be used for optimization.

After partitioning and sorting the partition rows, the resulting set is buffered again using the Record Buffering access method, on top of which the Window Partition partitioning method will be used.

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

Once all partitions are defined and the rows within the partitions are ordered, the computation of the window functions themselves begins, which is displayed in the Explain plan as Window. During the computation of window functions within a partition, the set of rows for the function computation can either grow from the start of the partition (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW), or shrink when moving towards the end of the partition (ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING), or have a fixed size moving along the partition relative to the current row (ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING), or remain unchanged if the window frame is defined as ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING). This is why this access method is called “Sliding Window”.

In the Legacy plan, the computation of window functions is not displayed in any way, but if external sorting (SORT) is used for partitioning the window or ordering rows within a partition, it will be displayed.

Example 51. Computation of a Simple 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
```

Here, there is only one partition, which includes all records from the query.

Example 52. Computation of a 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
```

Let's add a row processing order within the partition, which will also implicitly set the window frame (`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`).

Example 53. Computation of a Partitioned Window Function with Row Processing Order Specified

```
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 multiple window functions with different windows are used.

Example 54. Computation of 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
```

3.4. Merging Methods

This category of access methods largely resembles filters. Merging methods also transform input data according to a specific algorithm. The formal difference is only that this group of methods always operates on several input streams. Furthermore, their usual result is either an expansion of the selection by fields or an increase in its cardinality.

There are two classes of merging methods — join and union, which perform different functions. In addition, the execution of recursive queries, which is a special case of unions, also belongs to this category. Below we will familiarize ourselves with their description and possible implementation algorithms.

3.4.1. Joins

As one might guess from the name, this group of methods implements SQL joins. The SQL standard defines two types of joins: inner and outer. Furthermore, outer joins are divided into one-sided (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-sided outer join, one stream is the leading (mandatory) one, and the second is the driven (optional) one. Often the leading stream is also called the outer stream, and the driven stream is called the inner stream. For a left outer join, the leading (outer) stream is the left one, and the driven (inner) stream is the right one. For a right outer join — it is the opposite.

I will immediately note that formally, a left outer join is equivalent to an inverted right outer join. Inversion in this context means swapping the outer and inner streams. So it is sufficient to implement only one type of one-sided outer join (usually the left one). This is how it is done in Firebird. However, the decision to transform a right join into a left (base) 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 joins at the BLR parsing level, which eliminates possible conflicts in the optimizer.

Besides input streams, each join has another attribute — the join condition. It is this condition that determines the result, i.e., how exactly the data from the input streams 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-sided outer joins. Their streams are not called leading and driven for nothing. In this case, the outer (leading) stream must always be read before the inner (driven) stream; otherwise, it will be impossible to perform the NULL value substitution required by the standard in case of missing matches from the inner stream to the outer one. From this, we can conclude that the optimizer does not have the ability to choose the execution order of a one-sided outer join, and it will always be determined by the query text. Whereas for inner and full outer joins, the input streams are independent and can be read in any order; therefore, the execution algorithm for such joins is determined solely by the optimizer and does not depend on the query text.

In addition to the joins presented in the SQL standard, many DBMSs also implement additional join methods: semi-join and anti-join. These types of joins are not directly represented in SQL but can be used by the optimizer after transforming the SQL query execution tree. For example, they can be used to execute the `EXISTS` and `NOT EXISTS` filter predicates. These types of joins are asymmetric by nature, meaning the joined streams are not equivalent for them — a leading (mandatory) stream and a driven stream (the one being checked in the filter predicate) are distinguished.

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 (the outer one) is opened and one record is read from it. After that, the second stream (the inner one) 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. Next, a second record is read from the inner stream and the process repeats until EOF is returned from it. If the inner stream does not contain any record matching the outer stream, then two scenarios are possible. In the case of an inner join, the record is not returned as output. In the case of an outer join, we join the record from the outer stream with the necessary number of NULL values and output it. Then, regardless of the join type, we read the second record from the outer stream and start the iteration process over the inner stream again. It is obvious that this algorithm works on a pipeline principle and the joining of streams is performed directly during the client fetch.

A 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, the nested loop join works efficiently only if there is an index applicable to the join condition. In this case, on each iteration, a subset of records satisfying 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, only the one corresponding to the join condition. For example, with a join condition like (`ON SLAVE.F = 0`), even when using an index on the `F` field of the inner table, the same records will still be read from it on each iteration, which is a waste of resources. In this

situation, a merge join would be more efficient (see below).

One can introduce the definition of "stream dependency" as the presence of fields from both streams in the join condition and conclude that the nested loop join is efficient only when the streams are dependent on each other.

If we recall the description of "Predicate Checking", which describes the principle of the optimizer placing predicate filters as "deeply" as possible, then one point becomes clear: individual table filters will go "below" the join methods. Accordingly, in the case of a predicate like (`WHERE MASTER.F = 0`) and the absence of records with the field `F` equal to zero in the `MASTER` table, there will be no accesses to the inner stream of the join at all, because in this case, there is no iteration over the outer stream (it has no records).

Since the logical connectives `AND` and `OR` are optimized via bitmaps, the nested loop join can be used for all types of join conditions, and usually quite efficiently.

For outer joins, the nested loop join is always chosen, since currently there are no alternative options. For inner joins, the optimizer chooses the nested loop join if its cost is cheaper than alternative options (Hash Join). If there are no suitable indexes for the join fields of inner joins, then alternative join algorithms are almost always chosen, if they are possible. Furthermore, for inner joins, the optimizer chooses the most efficient order for joining the streams. The main selection criteria are the cardinalities of both streams and the selectivity of the join condition. The following formulas are used to estimate the cost of a specific 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 determines the cost of selecting from the inner stream on each iteration. Multiplying it by the number of iterations gives the total cost of selecting from the inner stream. The total cost is obtained by adding the cost of selecting from the outer stream. From all possible permutations, the option with the lowest cost is selected. During the permutation process, obviously worse options are discarded (based on the already available cost information). Not only streams joined by nested loops (Nested Loop) participate in the permutations, but also streams joined by other join algorithms (Hash Join), whose cost is calculated by different rules.

In the Legacy execution plan, streams joined by nested loops are displayed with the word "JOIN", followed by the input streams described in parentheses separated by commas.

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

Nested Loop Join (inner)

Example 55. 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)
```

Let's note one point. Since all streams are equivalent for an inner join, the optimizer converts the binary tree into a "flat" form when there are more than two streams. This means 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 56. 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 also belongs to inner joins and is always performed using nested loops.

Example 57. 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 58. 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 59. 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 an inner join, the Explain plan here contains nested “Nested Loop Join”.

Nested Loop Join (semi)

Currently, this type of join is 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. Then 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, the record from the outer stream is output, and the inner stream is immediately closed. If the join condition is not met, then the second record is read from the inner stream and the process repeats until EOF is returned from it. Then the outer stream reads the next record and everything repeats until EOF is returned from it.

This access method can be used to execute subqueries in the `EXISTS` and `IN (<select>)` predicates, as well as other predicates that are transformed into `EXISTS` (`ANY`, `SOME`).

Estimating the cardinality of the output stream is very difficult, but what can be said for sure is that it never exceeds the cardinality of the outer stream. It is assumed that half of the records from the outer stream satisfy 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 fetching the first record that matches the join condition. Since the `EXISTS` predicate can contain a complex subquery, and not just reading from a single table, calculating it is not so simple. It is also quite obvious that for this algorithm to work efficiently, the optimizer must switch to the `FIRST ROWS` optimization strategy for executing the inner subquery.

Nested Loop Join (anti)

Currently, this type of join is not used by the Firebird optimizer.

Essentially, 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. Then 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 met, the record from the outer stream is output, and the inner stream is immediately closed. If the join condition is met, then the second record is read from the inner stream and the process repeats until EOF is returned from it. Then the outer stream reads the next record and everything repeats until EOF is returned from it.

This access method can be used to execute subqueries in the `NOT EXISTS` predicate, as well as other predicates that are transformed into `NOT EXISTS` (`ALL`). Furthermore, under certain conditions, some one-sided outer queries can also be transformed into it.

Estimating the cardinality of the output stream is very difficult, but what can be said for sure is that it never exceeds the cardinality of the outer stream. It is assumed that half of the records from the outer stream do not satisfy 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 fetching the first record that matches the join condition. Since the `NOT EXISTS` predicate can contain a complex subquery, and not just reading from a single table, calculating it is not so simple. It is also quite obvious that for this algorithm to work efficiently, the optimizer must switch to the `FIRST ROWS` optimization strategy for executing the inner subquery.

Hash Join

Hash join is an alternative algorithm for performing joins. Stream joining using the HASH JOIN algorithm has been available since Firebird 3.0.

In a hash join, the input streams are always divided into a leading and a trailing stream, with the trailing stream typically being the one with the lower cardinality. First, the smaller (trailing) stream is read entirely into an internal buffer. During the reading process, a hash function is applied to each join key, and the pair {hash, buffer pointer} is written to a hash table. Then the leading stream is read, and its join key is probed in the hash table. If a match is found, the records from both streams are joined and output. In case of multiple duplicates of this key in the trailing table, multiple records will be output. If the key is not found in the hash table, the process moves to the next record in the leading stream, and so on.

Obviously, replacing key comparisons with hash comparisons is only possible when comparing for strict key equality. Therefore, a join with a join condition of the form (`ON MASTER.F > SLAVE.F`) cannot be executed using the Hash Join algorithm.

Nevertheless, this method has one advantage over nested loops join — hash join allows joining based on expressions. For example, a join with a condition of the form (`ON MASTER.F + 1 = SLAVE.F + 2`) is easily performed using the hash method. The reason is clear: there is no dependency between the streams and, consequently, no requirement to use indexes.

Collisions can occur during the construction of the hash table. A collision is a situation where different key values result in the same hash function value. Therefore, after a hash function value match, the join predicate is always recalculated. Collisions are stored as a list, sorted by keys, which allows for a binary search within the collision chain.

In the current implementation, the hash table has a fixed size of 1009 slots; it does not change based on the estimated cardinality of the hashed stream. Also, the hash table size does not increase, and rehashing does not occur when collision chain lengths are exceeded. This means that hash join is efficient only with a relatively small cardinality of the trailing stream. If the cardinality of the hashed stream exceeds 1,009,000 records, other join algorithms are selected (NESTED LOOP JOIN if indexes are available, otherwise MERGE JOIN).

This may change in future versions of Firebird.

Why 1,009,000 records? 1,009,000 / 1009 slots = 1000 possible collisions, searching through sorted collisions takes log2(1000) = 10 steps, which is already considered inefficient.

The hash join algorithm is capable of processing multiple join conditions combined with AND. In this case, the hash function is calculated for multiple fields included in the join condition (or multiple joins, if more than one stream is being joined). However, if join conditions are combined with OR, the hash join method cannot be applied. It should be noted here that the hash function is not always calculated for all fields in the join condition; it may be calculated only for a subset of them. This happens if the number of join keys exceeds 8. In this case, when the hash function value matches, more records than necessary will be returned from the hash table. The extra records will be filtered out after evaluating all predicates in the join condition. In this case, the efficiency of the hash join decreases. The Explain plan does not show the number of keys used 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 `NULL` values into account. That is, both `=` and `IS NOT DISTINCT FROM` are allowed. However,

when building the hash table, no distinction is made between these predicates, although it is obvious that if the = predicate is used, records with a NULL key can be omitted from the hash table. This leads to increased memory consumption and reduced performance of the hash join if there are many keys with NULL values. This shortcoming is planned to be fixed in future versions of Firebird (see CORE-7769).

Before Firebird 5.0, the HASH JOIN method was used only in the absence of indexes for the join condition or their inapplicability; otherwise, the optimizer chose the nested loops join algorithm (NESTED LOOP) using indexes. In fact, this is not always optimal. If a large stream is joined with a small table via the primary key, each record of that table will be read multiple times; furthermore, index pages will also be read multiple times if they are used. When using HASH 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 the HASH JOIN method consists of the following parts:

- the cost of retrieving 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 where hashes matched.

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

```
// output stream cardinality
cardinality = outerCardinality * innerCardinality * linkSelectivity

// hash table cardinality, if not all join condition fields are hash table keys
hashCardinality = innerCardinality * outerCardinality * hashSelectivity

// if all join condition fields are hash table keys
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` — the selectivity of the join keys for which the hash function was calculated;
- `linkSelectivity` — the selectivity of the join condition;
- `innerCost` — the cost of retrieving records for the hashed stream;
- `innerCardinality` — the cardinality of the trailing (hashed) stream;

- `outerCardinality` — the cardinality of the leading stream;
- `COST_FACTOR_MEMCOPY` - the cost of copying a record to/from memory;
- `COST_FACTOR_HASHING` - the cost of calculating the hash function.

In the Legacy plan, a hash join is represented by the word "HASH", followed by the input streams described in parentheses, separated by commas.

In the Explain plan, a hash join is represented as a tree, whose root is labeled “Hash Join” followed by the join type in parentheses. The joined streams are described one level below. Hashing of the trailing table is displayed as “Record Buffer”, followed by the record length as `(record length: <length>)` in parentheses (see Record Buffering).

Hash Join (inner)

Example 60. 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-sided outer hash join is not implemented in Firebird. Its execution algorithm is similar to that of an inner join, with one exception — the inner (driven or optional stream) is always hashed. For each record from the outer (mandatory) stream, the hash table is probed; if a match is found, the two records are merged into one and output. If no match is found, the record from the outer stream is output with the necessary number of NULL values.

Hash Join (semi)

This access method can be used to execute subqueries in `EXISTS` and `IN (<select>)` predicates, as

well as other predicates that are transformed into EXISTS (ANY, SOME). It has been available since Firebird 5.0.1. This feature is disabled by default; to enable it, you must set the SubQueryConversion configuration parameter to true in the firebird.conf or database.conf file.

Not every subquery in EXISTS can be converted to a semi-join. If the subquery contains FETCH/FIRST/SKIP/ROWS limiters, the subquery cannot be converted to a semi-join and it will be executed as a regular correlated subquery. As with inner join, to perform a semi-join using hashing, the keys must be compared for strict equality; the use of expressions in the join conditions is allowed, and the join conditions can be combined with AND.

The algorithm for hash semi-join is as follows. The subquery stream is selected as the driven stream and is entirely read into an internal buffer. During reading, a hash function is applied to each join key, and the pair {hash, pointer in the buffer} is written to the hash table. After that, the driving stream is read and its join key is probed in the hash table. If a match is found, the record from the outer stream is output. If the key is not found in the hash table, we move to the next record of the driving stream, and so on.

Estimating the cardinality of the output stream is quite difficult, but what can be said for sure is that it never exceeds the cardinality of the outer stream. It is assumed that half of the records from the outer stream satisfy the join condition.

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

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

Example 61. 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 join condition is `P.RDB$RELATION_ID = R.RDB$RELATION_ID`. This query is **as if** 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 shown for a 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 EXISTS` predicate, as well as other predicates that are transformed into `NOT EXISTS` (`ALL`). Furthermore, under certain conditions, some one-sided outer queries can also be transformed into it. In current versions of Firebird, it is not implemented.

The algorithm for hash anti-join is as follows. The subquery stream is selected as the driven stream and is entirely read into an internal buffer. During reading, a hash function is applied to each join key, and the pair {hash, pointer in the buffer} is written to the hash table. After that, the driving stream is read and its join key is probed in the hash table. If no match is found, the record from the outer stream is output. If a match is found, we move to the next record of the driving stream, and so on.

Single-Pass Merge (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-sorted by the join key, after which a binary merge tree is built. Then, during fetching, records are read from both streams and their join keys are compared for equality. If the keys match, the record is output. Then new records are read from the input streams and the process repeats. The merge is always performed in a single pass, meaning each input stream is read only once. This is possible due to the ordered arrangement of the join keys in the input streams.

However, this algorithm cannot be used for all types of joins. As noted above, it requires a strict equality comparison of keys. Thus, a join with a join condition of the form (`ON MASTER.F > SLAVE.F`) cannot be performed via a single-pass merge. To be fair, joins of this kind cannot be efficiently performed by any method, because even when using the nested loops join algorithm, repeated reads from the inner stream will occur on each iteration.

In some DBMSs, for example Oracle, merge join is possible not only for strict equality, meaning joins with conditions of the form (`ON MASTER.F > SLAVE.F`) are possible.

Nevertheless, this method has one advantage compared to the nested loops join algorithm — merging based on expressions is allowed. For example, a join with a condition of the form (`ON MASTER.F + 1 = SLAVE.F + 2`) is easily performed by the merge method. The reason is clear: there is no dependency between the streams and, consequently, 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 for the case of stream dependency (one-sided outer join). The answer is simple: this algorithm is not supported for such joins. Furthermore, it is also not supported for full outer joins, semi-joins, and anti-joins.

Support for outer joins by the algorithm is planned for future versions of the server.

This algorithm is capable of processing several join conditions combined with AND. In this case, the input streams are simply sorted by several fields. However, if the join conditions are combined with OR, the merge method cannot be applied.

The main disadvantage of this algorithm is the requirement to sort both streams by the join keys. If the streams are already sorted, then this join algorithm is the cheapest among others (NESTED LOOP, HASH JOIN). However, usually the joined streams are not sorted by the join keys, and therefore they have to be sorted, which sharply 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 the join keys, and therefore this algorithm might not be used in cases where it would actually be cheaper.

This shortcoming is planned to be fixed in future versions of the server.

Before Firebird 3.0, merge joins were used only in cases where using the nested loops join algorithm was impossible or suboptimal, primarily in the absence of indexes on the join condition or their inapplicability, as well as in the absence of 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 in cases where the joined streams are too large and hash join becomes inefficient (the cardinality of all joined streams is greater than 1009000 records, see Hash Join), as well as in the absence of indexes on the join condition, which would also make the nested loops join algorithm suboptimal.

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

```
cost = cost_1 + cost_2

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

Where `cost_1`, `cost_2` are the costs of retrieving data from the sorted input streams. And since the

cost of external sorting is not calculated, the cost of the merge join cannot be correctly estimated either.

In the Legacy execution plan, a merge join is indicated by the word "MERGE", followed by the input streams described in parentheses, separated by commas.

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

Example 62. 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 join using the merge method.

Full Outer Join

This type of join is not executed directly; instead, it is decomposed into an equivalent form — first, one stream is joined with 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 the streams in a full outer join are semantically equivalent, the optimizer can swap them depending on which is cheaper.

Example 63. 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 version 3.0, it could not use indexes for a full outer join, which made this type of join extremely inefficient. A way out of this situation was to rewrite 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. Now the optimizer performs such transformations for you automatically.

Join 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 stored procedure does not depend on other streams, then the order and algorithm of the join for inner joins (INNER JOIN) depend on which join predicate is used and whether it is possible to use an index on the table's field (or expression for the field).
  - An equality predicate (or several equality predicates combined with `AND`) is used and there is no index on the table's field (or expression) — the `HASH JOIN` algorithm is used. The driving stream becomes the one with the lower cardinality. For stored procedures, the cardinality is assumed to be 1000;
  - A predicate other than `=` or `IS NOT DISTINCT FROM` is used, or an index on the table's field (or expression) exists and can be applied — the `Nested Loop Join` algorithm is used, and the stored procedure becomes the driving stream. This allows it to be executed once instead of being re-executed on every iteration (since we cannot apply an index to it).
- If the stored procedure is joined using one-sided outer joins (LEFT/RIGHT JOIN), then the join order is determined by the order of the table and the procedure.
- If the stored procedure depends on other streams (through input parameters), then it becomes the driven stream for any type of join. If the join order cannot be rearranged in this case (`table RIGHT JOIN proc(table.field)`), an error occurs.

Example 64. Join with a non-correlated 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
```

Now let's change the query text so that it is not possible to use the PK_HORSE index.

```
SELECT *
FROM SP_PEDIGREE(?) P
JOIN HORSE ON HORSE.CODE_HORSE+0 = P.CODE_HORSE
```

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

```
[cardinality=585403.4, cost=???]
Select Expression
    [cardinality=585403.4, cost=???]
    -> Filter
        [cardinality=585403.4, cost=???]
        -> Hash Join (inner)
            [cardinality=585403.4, cost=585403.4]
            -> Table "HORSE" Full Scan
            [cardinality=1000.0, cost=????]
            -> Record Buffer (record length: 137)
                [cardinality=1000.0, cost=????]
                -> Procedure "SP_PEDIGREE" as "P" Scan
```

In this case, the cardinality estimate for the HORSE table turned out to be greater than the cardinality estimate for the stored procedure, so the table was chosen as the driving stream, and the results of the stored procedure execution are hashed.

If the procedure's input parameters depend on other streams, the optimizer detects these dependencies and sorts the streams so that the procedure becomes driven by the input streams.

Example 65. 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 dependency of the procedure's input parameters on other streams. In this case, the procedure was placed first, when no record had yet been selected from the table. Accordingly, at the execution stage, an `isc_no_cur_rec` error occurred ("no current record for fetch operation"). To work around this problem, the join order was explicitly specified 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.

Join 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 or CTE for short.

A derived table is a table expression included in the `FROM` clause of a query. Derived tables are queries in parentheses. This query can be given an alias, after which the table expression can be referenced like a regular table (used as a data source).

Example 66. Query using a 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 or CTE for short is a named table expression declared in the `WITH` clause. A common table expression, just like a derived table, can be referenced like a regular table (used as a data source). Common table expressions declared later can use earlier declared table expressions in their body. Common table expressions are divided into recursive and non-recursive. We will discuss 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 a CTE as follows:

Example 67. Query using a 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 content 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 base tables. In more complex cases, the table expression cannot be expanded to base tables. Simple table expressions contain only inner join operations on streams and their filter conditions. Adding sorting, `DISTINCT`, aggregate computation, grouping, window function computation, `UNION`, and FIRST/SKIP counters, and outer joins turns the table expression into a complex one.

The optimizer behaves differently for inner and outer joins with complex table expressions. In the case of an inner join of a table with a table expression, the optimizer tries to choose an execution plan that would not lead to re-execution of the query inside the table expression. Thus, the join of the table expression with the table is performed using one of two algorithms:

- `Nested Loop Join`. In this case, the table expression is chosen as the driving stream, and the table as the driven stream. This algorithm is chosen only if an index on the table field cannot be used for the join predicate or if the join predicate cannot be used for the `Hash Join` algorithm.

- `Hash Join`. Used if the join predicate is an equality (`=`) or equivalence (`IS NOT DISTINCT FROM`), and there is no possibility to use an index on the table's field(s). In this case, the cardinality of the table expression and the table is estimated. The data stream with the lower cardinality is hashed and becomes the driven stream. In both cases, the query inside the table expression is optimized separately, meaning its plan does not depend on what it is being joined with. Filtering predicates that are independent of other streams and are applied to the table expression will, where possible, be pushed down into the table expression.

For one-sided outer joins, the join order is always determined by the query text, and the join itself is performed by the `Nested Loop Join` algorithm. If a complex table expression is selected as the driven stream, it will be re-executed for each record of the driving stream. In this case, predicates applicable to the table expression, including the join condition, will, where possible, be pushed down into the table expression.

Let's look at an example of a join with a simple table expression.

Example 68. Expanding CTE 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 base tables and optimization occurred 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 69. 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 drastically. First, the query inside the CTE was executed, and only then were the other streams joined to it. This plan was chosen because an index exists for the RDB$RELATION_NAME field of the RDB$RELATIONS table. The join order will not change regardless of the ratio of record counts in the table and the table expression.

However, if there is no suitable index for efficient execution of the join using the Nested Loop Join method and it is possible to perform this join using the Hash Join algorithm, then the latter will be chosen. If the Hash Join algorithm is selected, the cardinality of the table and the table expression is estimated, and the smallest stream will be chosen for hashing and will become the driven stream.

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

Select Expression
    -> Filter
        [cardinality=799.0, cost=18386.2]
        -> Hash Join (inner) (keys: 1, total key length: 252)
            [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$INDEX_2" Range Scan (full match)
        [cardinality=322.9, cost=645.8]
        -> Record Buffer (record length: 273)
            [cardinality=322.9, cost=322.9]
            -> Table "RDB$RELATIONS" as "R" Full Scan
```

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

Example 70. Complex CTEs, joined with `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 that are independent of outer streams, the standard provides for derived tables that are dependent on outer streams. To use such derived tables, it is necessary to use the keyword `LATERAL` before the table expression. Common table expressions 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 for other types of joins. A feature of joining with a lateral derived table is that they can only be executed by a nested loop algorithm. Another feature concerns the execution of `CROSS JOIN`. The fact is that because the LATERAL derived table depends on outer streams, it cannot be set as the driving stream in joins. The optimizer recognizes the presence of dependencies and selects the correct join order.

Example 71. 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)
```

Joining with Views

A view is a virtual (logical) table, which is a named query (a synonym for a query) that will be substituted as a table expression when the view is used.

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

3.4.2. Unions (Union)

The name of the access method speaks for itself. This access method performs a SQL union operation. There are two modes for executing 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 case of `DISTINCT`, it is necessary to eliminate full duplicate records present in the union result. For this purpose, a sort filter is placed at the output of the union method, operating in a "truncating" mode on all fields.

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

In the Legacy plan, the union is represented by separate plans for each input stream.

In the Explain plan, the union is represented as a tree, whose root is labeled "Union". The level below describes the streams being unioned.

Example 72. Union of streams in ALL mode

```
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 73. Union of streams in DISTINCT mode

```
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
```

Materialization of Non-Deterministic Expressions

Table expressions in Firebird have one unpleasant feature, namely, when accessing columns of a table expression that reference expressions, these expressions are evaluated every time the table

expression's column is mentioned. This is easily demonstrated using non-deterministic functions.

Try executing 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
```

Even though you expected the values in the columns to be the same.

This also manifests when sorting by a reference to a column. For example:

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

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

There is one trick that allows you to "materialize" the results of expression evaluations in table expressions. To do this, it is sufficient to perform a `UNION` with a query that returns 0 records, naturally the total number of columns in both queries must be the same. Let's try rewriting 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 ID1 and ID2 will be the same.

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

Materialization of Subqueries

What happens if a subquery is used as a column of a table expression? The subquery can be heavy enough that you wouldn't want to execute it 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 only once for each record from `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" denotes the materialization of the subquery results for each record from `RDB$RELATIONS`. This is not some separate access method, but an internal union (UNION) within the table expression with an empty set. That is, the same thing happened as described above regarding the materialization of non-deterministic expressions.

3.4.3. Recursion

This access method is used to execute recursive table expressions (recursive CTEs). The body of a recursive CTE is a query with `UNION ALL`, which combines one or more subqueries called anchor

members. In addition to the anchor members, there are one or more recursive subqueries, called recursive members. These recursive subqueries reference the recursive CTE itself. So, we have one or more anchor subqueries and one or more recursive subqueries, combined with `UNION ALL`.

The essence of recursion is simple — first, the non-recursive subqueries are executed and combined, and for each record from the non-recursive part, the dataset 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.

When executing recursive CTEs, there is another feature — no predicates from the outer query can be pushed down into the CTE.

Estimating the cardinality and cost of recursion is quite difficult. The optimizer considers the cardinality to be equal to 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 from the cost of fetching all 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 plan, recursion is not displayed.

In the Explain plan, recursion is displayed as a tree, whose root is labeled “Recursion”. The level below describes the streams being combined. The reference to the recursive CTE itself within the CTE is not described as a separate data stream.

Example 74. 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 performs "infinite" recursion. But that’s in theory; in practice, Firebird limits the recursion depth to 1024. Let’s fix this.

Example 75. 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, the recursive part becomes visible, which is filtered, but the stream itself is not shown. Since the recursive member does not join with other data sources, the cardinality and cost are taken from the non-recursive part. 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 joins) with other data sources.

Example 76. Recursive query with a join on a 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 in the recursive part of the query, a join with the DEPARTMENT table is performed via 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 reality, 21 records will be selected.

Let's try to unfold the recursion.

Example 77. Recursive query with a join via a 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 in the recursive part of the query, a join with the DEPARTMENT table is performed via a unique index, so the overall cardinality does not increase (for 1 record from the non-recursive member, 1 record from the recursive member is joined). In practice, 4 records will be returned.

3.5. Optimization Strategies

While examining Firebird data access methods, we found that data sources can be pipelined and buffered. A pipelined data source emits records while reading from its input streams, whereas a buffered source must first read all records from its input streams and only then can it emit the first record to its output.

Most access methods are pipelined. Buffered access methods include external sorting (SORT) and record buffering (Record Buffer). Furthermore, some access methods require the use of buffered data sources. Hash Join requires buffering when building the hash table. Merge Join requires the input streams to be ordered by the join keys, for which external sorting is used. Computing window functions (Window) also requires buffering.

In Firebird, the optimizer can operate in two modes:

- FIRST ROWS — the optimizer builds a query plan to retrieve only the first rows of the query as quickly as possible;
- ALL ROWS — the optimizer builds a query plan to retrieve 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 that display only the first rows of the result, and the rest are fetched as needed, then the FIRST ROWS strategy might be more preferable, as it reduces the response time.

When using the FIRST ROWS strategy, the optimizer tries to replace buffered access methods with alternative pipelined ones, if it is possible and cheaper in terms of the cost of retrieving the first row. That is, Hash/Merge joins are 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. Furthermore, using first/skip counters in the query implicitly switches the optimization strategy to `FIRST ROWS`. Let's compare query plans that use different optimization strategies.

Example 78. 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 79. 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
```