Typical SQL Query Optimization Mistakes for the Firebird DBMS Part 1

Alexey Kovyazin, Firebird Foundation

Denis Simonov, IBSurgeon

MAIN MISTAKES

1. MEASURING TIME WITHOUT FETCH ALL

- Occurs when the query is a cursor and returns more than 100 rows
- You need to understand the optimization goal (FIRST ROWS / ALL ROWS)
- A fast return of the first rows does not mean fast retrieval of all rows
- This leads you to subconsciously optimize the query for the FIRST ROWS strategy (buffering access methods like SORT and HASH JOIN are avoided)
- If you only need the first rows quickly, specify OPTIMIZE FOR FIRST ROWS directly in the SQL
- An alternative to FETCH ALL is transforming the original query:

```
SELECT COUNT(*) FROM (<your query>)
```

2. PREMATURE OPTIMIZATION

- The SQL query is written right away according to your own idea of optimal execution
  - Forcing join order using LEFT JOIN
  - Sprinkling in +0 and ||'' hints
  - Creating composite indexes and expression indexes (mistakes with plain indexes are rare)
- How to do it correctly:
  - SQL is a declarative language. First write the query to match the required semantics, and only start optimizing if it performs poorly

3. READING EVERYTHING USING INDEXES

```
SELECT COUNT(*)
FROM
  HORSE
  JOIN BREED ON BREED.CODE_BREED = HORSE.CODE_BREED
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
  JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
WHERE HORSE.CODE_DEPARTURE = 1;

PLAN HASH (HASH (JOIN (COLOR NATURAL, HORSE INDEX (FK_HORSE_COLOR, FK_HORSE_DEPARTURE)), BREED NATURAL), FARM
NATURAL)

Per table statistics:
--------------------------------+---------+---------+
 Table name                     | Natural | Index   |
--------------------------------+---------+---------+
BREED                           |     290 |         |
COLOR                           |     242 |         |
FARM                            |   41356 |         |
HORSE                           |         |  100186 |
--------------------------------+---------+---------+
```

Default plan from 5.0

3. READING EVERYTHING USING INDEXES (CONTINUED)

```
Select Expression
    -> Aggregate
        -> Filter
            -> Hash Join (inner)
                -> Hash Join (inner)
                    -> Nested Loop Join (inner)
                        -> Table "COLOR" Full Scan
                        -> Filter
                            -> Table "HORSE" Access By ID
                                -> Bitmap And
                                    -> Bitmap
                                        -> Index "FK_HORSE_COLOR" Range Scan (full match)
                                    -> Bitmap
                                        -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)
                    -> Record Buffer (record length: 25)
                        -> Table "BREED" Full Scan
                -> Record Buffer (record length: 33)
                    -> Table "FARM" Full Scan
```

```
Current memory = 554636400
Delta memory = 352
Max memory = 557911200
Elapsed time = 0.692 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 177746
```

3. READING EVERYTHING USING INDEXES (MODIFIED)

```
SELECT COUNT(*)
FROM
  HORSE
  JOIN BREED ON BREED.CODE_BREED = HORSE.CODE_BREED
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
  JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
WHERE HORSE.CODE_DEPARTURE = 1
PLAN JOIN (HORSE INDEX (FK_HORSE_DEPARTURE), COLOR INDEX (PK_COLOR), BREED INDEX (PK_BREED),
  FARM INDEX (PK_FARM));

Per table statistics:
------------------------------+---------+---------+
 Table name                   | Natural | Index   |
------------------------------+---------+---------+
BREED                         |         |  100186 |
COLOR                         |         |  100186 |
FARM                          |         |  100186 |
HORSE                         |         |  100186 |
------------------------------+---------+---------+
```

Manual plan from 2.5

3. READING EVERYTHING USING INDEXES (CONTINUED)

```
Select Expression
    -> Aggregate
        -> Nested Loop Join (inner)
            -> Filter
                -> Table "HORSE" Access By ID
                    -> Bitmap
                        -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)
            -> Filter
                -> Table "COLOR" Access By ID
                    -> Bitmap
                        -> Index "PK_COLOR" Unique Scan
            -> Filter
                -> Table "BREED" Access By ID
                    -> Bitmap
                        -> Index "PK_BREED" Unique Scan
            -> Filter
                -> Table "FARM" Access By ID
                    -> Bitmap
                        -> Index "PK_FARM" Unique Scan
```

```
Current memory = 554878448
Delta memory = 464
Max memory = 557911200
Elapsed time = 0.417 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 1117200
```

3. WHAT SHOULD ACTUALLY HAVE BEEN DONE?

```
SELECT COUNT(*)
FROM
  HORSE
  JOIN BREED ON BREED.CODE_BREED = HORSE.CODE_BREED
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR+0
  JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
WHERE HORSE.CODE_DEPARTURE = 1;

PLAN HASH (HASH (HASH (HORSE INDEX (FK_HORSE_DEPARTURE), COLOR NATURAL), BREED NATURAL), FARM NATURAL

Per table statistics:
--------------------------------+---------+---------+
 Table name                     | Natural | Index   |
--------------------------------+---------+---------+
BREED                           |     290 |         |
COLOR                           |     242 |         |
FARM                            |   41356 |         |
HORSE                           |         |  100186 |
--------------------------------+---------+---------+
```

Plan from 5.0 with tweak

3. WHAT SHOULD ACTUALLY HAVE BEEN DONE? (CONTINUED)

```
Select Expression
    -> Aggregate
        -> Filter
            -> Hash Join (inner)
                -> Hash Join (inner)
                    -> Hash Join (inner)
                        -> Filter
                            -> Table "HORSE" Access By ID
                                -> Bitmap
                                    -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)
                        -> Record Buffer (record length: 25)
                            -> Table "COLOR" Full Scan
                    -> Record Buffer (record length: 25)
                        -> Table "BREED" Full Scan
                -> Record Buffer (record length: 33)
                    -> Table "FARM" Full Scan
```

```
Current memory = 555176256
Delta memory = 352
Max memory = 558199840
Elapsed time = 0.157 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 154439
```

4. MULTIPLE INDEXES INSTEAD OF A COMPOSITE ONE

- Why is a composite index better?
  - Firebird treats predicates as independent, so for predicates joined by AND the selectivities get multiplied. This is often not actually true — the result is an incorrect cardinality estimate for a table read using several indexes
  - A search using one index is slightly cheaper than a search using several
- Where a composite index is worse
  - It is not always possible to use it efficiently (engaging all segments)
  - Single-column indexes plus a composite for the combination slow down table modification (extra writes for index pages) and slow down garbage collection on the table

4. WHEN IS A COMPOSITE INDEX OF LITTLE USE?

- Only one segment of the composite index is used

```
CREATE INDEX IDX_SOME ON T(A, B);
SELECT * FROM T WHERE A > 0 AND B > 0;
```

- Or none at all

```
SELECT * FROM T WHERE B = 0;
```

If this is a query against a single table, the benefit of a composite index compared to several indexes is negligible. In such cases the index pages are read once, a bitmap is built, and then the data is retrieved using it. An incorrect cardinality estimate has no effect.

5. INCORRECT CARDINALITY ESTIMATE WITH MULTIPLE INDEXES

```
SELECT COUNT(*)
FROM
  HORSE
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
  JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
WHERE HORSE.CODE_DEPARTURE = 1
  AND HORSE.CODE_BREED = 65
  AND FARM.CODE_COUNTRY = 1;

PLAN JOIN (HORSE INDEX (FK_HORSE_BREED, FK_HORSE_DEPARTURE), COLOR INDEX (PK_COLOR),
  FARM INDEX (PK_FARM))

Per table statistics:
------------------------------+---------+---------+
 Table name                   | Natural | Index   |
------------------------------+---------+---------+
COLOR                         |         |   99982 |
FARM                          |         |   99982 |
HORSE                         |         |   99982 |
------------------------------+---------+---------+
```

5. INCORRECT CARDINALITY ESTIMATE (CONTINUED)

```
Select Expression
    -> Aggregate
        -> Nested Loop Join (inner)
            -> Filter
                -> Table "HORSE" Access By ID
                    -> Bitmap And
                        -> Bitmap
                            -> Index "FK_HORSE_BREED" Range Scan (full match)
                        -> Bitmap
                            -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)
            -> Filter
                -> Table "COLOR" Access By ID
                    -> Bitmap
                        -> Index "PK_COLOR" Unique Scan
            -> Filter
                -> Table "FARM" Access By ID
                    -> Bitmap
                        -> Index "PK_FARM" Unique Scan

            COUNT
=====================
            47868

Elapsed time = 0.317 sec   Fetches = 815046
```

5. INCORRECT CARDINALITY ESTIMATE (CONTINUED)

Stream cardinality after indexed access via CODE_DEPARTURE and CODE_BREED

Sel(FK_HORSE_BREED) = 0.003623188473284245
Sel(FK_HORSE_DEPARTURE) = 0.055555555596947670
Sel(bitmap and) 0.003623188473284245 * 0.055555555596947670 = 0.00020128825001550 57
CARD(HORSE) * Sel(bitmap and) = 562237 * 0.00020128825001550 57 = 113.17

This is smaller than in the other tables, so this stream is chosen as the driving stream, and joins with the remaining tables use the Nested Loop method.

5. MORE ACCURATE CARDINALITY ESTIMATE WITH A COMPOSITE INDEX

```
CREATE INDEX IDX_HORSE_BREED_DEP ON HORSE (CODE_DEPARTURE, CODE_BREED);

SELECT COUNT(*)
FROM
  HORSE
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
  JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
WHERE HORSE.CODE_DEPARTURE = 1
  AND HORSE.CODE_BREED = 65
  AND FARM.CODE_COUNTRY = 1;

PLAN HASH (HASH (HORSE INDEX (IDX_HORSE_BREED_DEP), COLOR NATURAL), FARM INDEX (FK_FARM_COUNTRY))

Per table statistics:
--------------------------------+---------+----------+
 Table name                     | Natural | Index    |
--------------------------------+---------+----------+
COLOR                           |     242 |          |
FARM                            |         |    36865 |
HORSE                           |         |    99982 |
--------------------------------+---------+----------+
```

5. MORE ACCURATE CARDINALITY ESTIMATE WITH A COMPOSITE INDEX

```
Select Expression
    -> Aggregate
        -> Filter
            -> Hash Join (inner)
                -> Hash Join (inner)
                    -> Filter
                        -> Table "HORSE" Access By ID
                            -> Bitmap
                                -> Index "IDX_HORSE_BREED_DEP" Range Scan (full match)
                    -> Record Buffer (record length: 25)
                        -> Table "COLOR" Full Scan
                -> Filter
                    -> Table "FARM" Access By ID
                        -> Bitmap
                            -> Index "FK_FARM_COUNTRY" Range Scan (full match)

            COUNT
=====================
            47868

Elapsed time = 0.121 sec    Fetches = 145418
```

5. MORE ACCURATE CARDINALITY ESTIMATE WITH A COMPOSITE INDEX

Stream cardinality after indexed access via the composite index

Sel(IDX_HORSE_BREED_DEP) = 0.00122549023944884 5
CARD(HORSE) * Sel = 562237 * 0.00122549023944884 5 = 689

This is greater than the cardinality of the COLOR table. The optimizer changed its estimate and chose HASH JOIN.

6. A COMPOSITE INDEX IS CHEAPER WITH REPEATED SCANNING

```
SELECT COUNT(*)
FROM
  HORSE
WHERE HORSE.CODE_MOTHER > 0
  AND HORSE.CODE_BREED = 65
  AND NOT EXISTS(
    SELECT * FROM COVER
    WHERE COVER.CODE_MOTHER = HORSE.CODE_MOTHER
      AND COVER.CODE_FATHER = 742363
  );

PLAN (COVER INDEX (FK_COVER_MOTHER, FK_COVER_FATHER))
PLAN (HORSE INDEX (FK_HORSE_BREED, FK_HORSE_MOTHER))

Per table statistics:
------------------------------+---------+---------+
 Table name                   | Natural | Index   |
------------------------------+---------+---------+
COVER                         |         |    1124 |
HORSE                         |         |  106948 |
------------------------------+---------+---------+
```

6. A COMPOSITE INDEX IS CHEAPER WITH REPEATED SCANNING

```
Sub-query
    -> Filter
        -> Table "COVER" Access By ID
            -> Bitmap And
                -> Bitmap
                    -> Index "FK_COVER_MOTHER" Range Scan (full match)
                -> Bitmap
                    -> Index "FK_COVER_FATHER" Range Scan (full match)
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" Access By ID
                -> Bitmap And
                    -> Bitmap
                        -> Index "FK_HORSE_BREED" Range Scan (full match)
                    -> Bitmap
                        -> Index "FK_HORSE_MOTHER" Range Scan (lower bound: 1/1)

            COUNT
=====================
            105824

Elapsed time = 0.939 sec    Fetches = 758223
```

6. A COMPOSITE INDEX IS CHEAPER WITH REPEATED SCANNING

```
CREATE INDEX IDX_COVER_FATHER_MOTHER ON COVER (CODE_FATHER, CODE_MOTHER);

PLAN (COVER INDEX (IDX_COVER_FATHER_MOTHER))
PLAN (HORSE INDEX (FK_HORSE_BREED, FK_HORSE_MOTHER))

Per table statistics:
------------------------------+---------+---------+
 Table name                   | Natural | Index   |
------------------------------+---------+---------+
COVER                         |         |    1124 |
HORSE                         |         |  106948 |
------------------------------+---------+---------+
```

6. A COMPOSITE INDEX IS CHEAPER WITH REPEATED SCANNING

```
Sub-query
    -> Filter
        -> Table "COVER" Access By ID
            -> Bitmap
                -> Index "IDX_COVER_FATHER_MOTHER" Range Scan (full match)
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" Access By ID
                -> Bitmap And
                    -> Bitmap
                        -> Index "FK_HORSE_BREED" Range Scan (full match)
                    -> Bitmap
                        -> Index "FK_HORSE_MOTHER" Range Scan (lower bound: 1/1)

           COUNT
====================
           105824

Elapsed time = 0.256 sec    Fetches = 544101
```

7. WRONG FIELD ORDER IN A COMPOSITE INDEX

- Only the first segment is used

```
CREATE INDEX IDX_COMPLEX ON WAYBILL (WAYBILL_DATE, SUPPLIER_ID);

SELECT ...
FROM WAYBILL ...
WHERE WAYBILL.SUPPLIER_ID = ?
  AND WAYBILL.BYDATE BETWEEN ? AND ?
```

7. WRONG FIELD ORDER IN A COMPOSITE INDEX

- Only the first segment is used

```
CREATE INDEX IDX_COMPLEX ON WAYBILL (SUPPLIER_ID, PAYED, BUYER_ID);

SELECT ...
FROM WAYBILL
WHERE SUPPLIER_ID = ? AND BUYER_ID = ?
```

End of part 1

- Questions? ak@firebirdsql.org
- Purchase book:
- https://firebirdsql.org/en/featured-products/book-secrets-of-firebird-sql-optimizer

Typical SQL Query Optimization Mistakes for the Firebird DBMS Part 2

Alexey Kovyazin, Firebird Foundation
Denis Simonov, IBSurgeon

Part 2

8. COMPUTING AGGREGATES WITH WIDE GROUPINGS

```
SELECT
  FARM.CODE_FARM,
  FARM.NAME,
  FARM.NAME_EN,
  FARM.ADDRESS,
  COUNT(*) AS CNT
FROM
  COVER
  JOIN FARM ON FARM.CODE_FARM = COVER.CODE_FARM
GROUP BY 1, 2, 3, 4;

Select Expression
    -> Aggregate
        -> Sort (record length: 2686, key length: 1828)
            -> Filter
                -> Hash Join (inner)
                    -> Table "COVER" Full Scan
                    -> Record Buffer (record length: 1441)
                        -> Table "FARM" Full Scan
```

8. COMPUTING AGGREGATES WITH WIDE GROUPINGS

```
Current memory = 554646368
Delta memory = 288
Max memory = 1092118992
Elapsed time = 11.670 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 846691
Per table statistics:
-------------------------------+---------+---------+---------+---------+---------+
 Table name                    | Natural | Index   | Insert  | Update  | Delete  |
-------------------------------+---------+---------+---------+---------+---------+
COVER                          |  767959 |         |         |         |         |
FARM                           |   41356 |         |         |         |         |
-------------------------------+---------+---------+---------+---------+---------+
```

8. COMPUTING AGGREGATES WITH WIDE GROUPINGS

- Keep in GROUP BY only the fields that uniquely identify the entity. For the rest, use MIN/MAX
- In Firebird 6.0, use the new ANY_VALUE aggregate function

```
SELECT
  FARM.CODE_FARM AS CODE_FARM,
  MIN(FARM.NAME) AS NAME,
  MIN(FARM.NAME_EN) AS NAME_EN,
  MIN(FARM.ADDRESS) AS ADDRESS,
  COUNT(*) AS CNT
FROM
  COVER
  JOIN FARM ON FARM.CODE_FARM = COVER.CODE_FARM
GROUP BY 1;

Select Expression
    -> Aggregate
        -> Filter
            -> Hash Join (inner)
                -> Table "COVER" Access By ID
                    -> Index "FK_COVER_FARM" Full Scan
                -> Record Buffer (record length: 1441)
                    -> Table "FARM" Full Scan
```

8. COMPUTING AGGREGATES WITH WIDE GROUPINGS

```
Current memory = 555886896
Delta memory = 320
Max memory = 1092118992
Elapsed time = 1.456 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 1745358
Per table statistics:
------------------------------+---------+---------+---------+---------+---------+
 Table name                   | Natural | Index   | Insert  | Update  | Delete  |
------------------------------+---------+---------+---------+---------+---------+
COVER                         |         |  767959 |         |         |         |
FARM                          |   41356 |         |         |         |         |
------------------------------+---------+---------+---------+---------+---------+
```

8. COMPUTING AGGREGATES WITH WIDE GROUPINGS 2

```
SELECT
  FARM.CODE_FARM,
  FARM.NAME,
  FARM.NAME_EN,
  FARM.ADDRESS,
  COUNT(*) AS CNT
FROM
  COVER
  LEFT JOIN FARM ON FARM.CODE_FARM = COVER.CODE_FARM
GROUP BY 1, 2, 3, 4;

Select Expression
    -> Aggregate
        -> Sort (record length: 2686, key length: 1828)
            -> Nested Loop Join (outer)
                -> Table "COVER" Full Scan
                -> Filter
                    -> Table "FARM" Access By ID
                        -> Bitmap
                            -> Index "PK_FARM" Unique Scan
```

8. COMPUTING AGGREGATES WITH WIDE GROUPINGS 2

```
Current memory = 551766656
Delta memory = 320
Max memory = 1028880208
Elapsed time = 9.265 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 3956383

Per table statistics:
------------------------------+---------+---------+---------+---------+---------+
 Table name                   | Natural | Index   | Insert  | Update  | Delete  |
------------------------------+---------+---------+---------+---------+---------+
COVER                         |  767959 |         |         |         |         |
FARM                          |         |  767959 |         |         |         |
------------------------------+---------+---------+---------+---------+---------+
```

8. COMPUTING AGGREGATES WITH WIDE GROUPINGS 2

- Transform into a semantically equivalent query

```
WITH
  S AS (
    SELECT COVER.CODE_FARM, COUNT(*) AS CNT
    FROM COVER
    GROUP BY 1
  )
SELECT
  FARM.CODE_FARM,
  FARM.NAME,
  FARM.NAME_EN,
  FARM.ADDRESS,
  S.CNT
FROM
  S LEFT JOIN FARM ON FARM.CODE_FARM = S.CODE_FARM;
```

```
Select Expression
    -> Nested Loop Join (outer)
        -> Aggregate
            -> Table "COVER" as "S COVER" Access By ID
                -> Index "FK_COVER_FARM" Full Scan
        -> Filter
            -> Table "FARM" Access By ID
                -> Bitmap
                    -> Index "PK_FARM" Unique Scan
```

8. COMPUTING AGGREGATES WITH WIDE GROUPINGS 2

```
Current memory = 552296832
Delta memory = 384
Max memory = 1028880208
Elapsed time = 0.593 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 1760461

Per table statistics:
------------------------------+---------+---------+---------+---------+---------+
 Table name                   | Natural | Index   | Insert  | Update  | Delete  |
------------------------------+---------+---------+---------+---------+---------+
COVER                         |         |  767959 |         |         |         |
FARM                          |         |   14758 |         |         |         |
------------------------------+---------+---------+---------+---------+---------+
```

9. ORDER VS SORT

- ORDER index – index navigation
  - Almost always beneficial when only the first rows of the table are retrieved
  - ORDER index on a large table provokes random reads if the page cache is insufficient (see the DefaultDbCachePages parameter)
- SORT – external sort
  - Cost depends on the cardinality and width of the result set
  - Can run in memory or use temporary files (see the TempCacheLimit parameter)
  - Sorting wide result sets can be optimized by the Refetch algorithm (see InlineSortThreshold, available since Firebird 4.0 / HQbird 3.0)
  - Refetch can be applied to ORDER BY or window functions, but not to DISTINCT/GROUP BY

9. ORDER AND NON-INDEXED FILTER PREDICATES

```
SELECT NAME
FROM HORSE
WHERE FAMILY = ''
ORDER BY NAME;

Select Expression
    -> Filter
        -> Table "HORSE" Access By ID
            -> Index "HORSE_IDX_NAME" Full Scan

PLAN (HORSE ORDER HORSE_IDX_NAME)

Elapsed time = 0.870 sec    Fetches = 1616407
```

9. ORDER AND NON-INDEXED FILTER PREDICATES

- The query returns only 3 rows. The non-indexed predicate has fairly good selectivity
- The filter is not indexed, which means the entire table is first read via index navigation, and only then is the filter applied
- **Very slow!**
- How to improve the situation:
  - Either create an index on FAMILY, or a composite (FAMILY, NAME). Not the other way around
  - Or disable index navigation

9. ORDER AND NON-INDEXED FILTER PREDICATES

```
SELECT NAME
FROM HORSE
WHERE FAMILY = ''
ORDER BY NAME || '';

PLAN SORT (HORSE NATURAL)

Select Expression
    -> Sort (record length: 556, key length: 304)
        -> Filter
            -> Table "HORSE" Full Scan

Elapsed time = 0.219 sec    Fetches = 589984
```

9. ORDER AND NON-INDEXED FILTER PREDICATES

- External sort is faster here because non-indexed filter predicates are applied before sorting
- If the filter predicate has poor selectivity, the situation is less clear-cut
- If searching for the first record (ORDER BY field FETCH FIRST ROW ONLY), non-indexed predicates are almost always harmful — it depends on luck whether the matching row comes first or last

10. REPETITIVE SUBQUERIES INSTEAD OF LATERAL

```sql
SELECT
    H.CODE_HORSE,
    H.NAME AS HORSE_NAME,
    (  SELECT R.CODE_FARM
       FROM REGISTRATION R
       WHERE R.CODE_HORSE = H.CODE_HORSE
         AND R.BYDATE <= DATE '2015-12-31'
       ORDER BY R.BYDATE DESC
       FETCH FIRST ROW ONLY
    ) AS CODE_FARM,
    (  SELECT R.CODE_REASON
       FROM REGISTRATION R
       WHERE R.CODE_HORSE = H.CODE_HORSE
         AND R.BYDATE <= DATE '2015-12-31'
       ORDER BY R.BYDATE DESC
       FETCH FIRST ROW ONLY
    ) AS CODE_REASON,
    (  SELECT R.BYDATE
       FROM REGISTRATION R
       WHERE R.CODE_HORSE = H.CODE_HORSE
         AND R.BYDATE <= DATE '2015-12-31'
       ORDER BY R.BYDATE DESC
       FETCH FIRST ROW ONLY
    ) AS BYDATE
FROM
  HORSE H
WHERE EXTRACT(YEAR FROM H.BIRTHDAY) = 2005;
```

10. REPETITIVE SUBQUERIES INSTEAD OF LATERAL

```
SELECT
  H.CODE_HORSE,
  H.NAME AS HORSE_NAME,
  LR.CODE_FARM,
  LR.CODE_REASON,
  LR.BYDATE
FROM
  HORSE H
  LEFT JOIN LATERAL (
    SELECT
      R.CODE_FARM,
      R.CODE_REASON,
      R.BYDATE
    FROM REGISTRATION R
    WHERE R.CODE_HORSE = H.CODE_HORSE
      AND R.BYDATE <= DATE '2015-12-31'
    ORDER BY R.BYDATE DESC
    FETCH FIRST ROW ONLY
  ) LR ON TRUE
WHERE EXTRACT(YEAR FROM H.BIRTHDAY) = 2005;
```

11. SUBQUERIES INSTEAD OF WINDOW FUNCTIONS

```
SELECT
  S.CODE_PAGE_CATEGORY AS CODE_PAGE_CATEGORY,
  100.0 * SUM(S.ELAPSED_TIME) /
    (SELECT SUM(SS.ELAPSED_TIME)
     FROM SERVICE SS
     WHERE SS.CODE_WEBUSER = 1
       AND SS.ELAPSED_TIME IS NOT NULL
    ) AS PERCENT_TIME
FROM
  SERVICE S
WHERE S.CODE_WEBUSER = 1
  AND S.ELAPSED_TIME IS NOT NULL
GROUP BY 1;

PLAN (SS INDEX (FK_SERVICE_REF_WEBUSER))
PLAN (S ORDER FK_SERVICE_REF_PAGE_CATEGORY INDEX (FK_SERVICE_REF_WEBUSER))
```

11. SUBQUERIES INSTEAD OF WINDOW FUNCTIONS

```
Current memory = 554748880
Delta memory = 464
Max memory = 557355616
Elapsed time = 1.137 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 989630
Per table statistics:
------------------------------+---------+---------+---------+---------+---------+
 Table name                   | Natural | Index   | Insert  | Update  | Delete  |
------------------------------+---------+---------+---------+---------+---------+
SERVICE                       |         |  804888 |         |         |         |
------------------------------+---------+---------+---------+---------+---------+
```

11. SUBQUERIES INSTEAD OF WINDOW FUNCTIONS

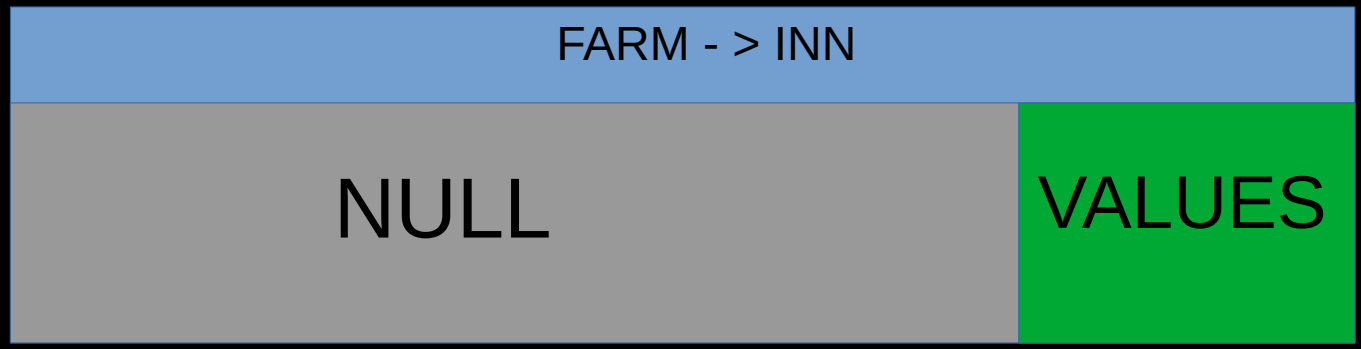
```
WITH
  T AS (
    SELECT
      DISTINCT
      SERVICE.CODE_PAGE_CATEGORY,
      SUM(SERVICE.ELAPSED_TIME) OVER() AS SUM_TIME,
      SUM(SERVICE.ELAPSED_TIME) OVER(PARTITION BY SERVICE.CODE_PAGE_CATEGORY)
        AS SUM_TIME_CAT
    FROM
      SERVICE
    WHERE SERVICE.CODE_WEBUSER = 1
      AND SERVICE.ELAPSED_TIME IS NOT NULL
  )
SELECT
  CODE_PAGE_CATEGORY,
  100.0 * SUM_TIME_CAT / SUM_TIME AS PERCENT
FROM T;

PLAN SORT (SORT (T SERVICE INDEX (FK_SERVICE_REF_WEBUSER)))
```

11. SUBQUERIES INSTEAD OF WINDOW FUNCTIONS

```
Current memory = 554956800
Delta memory = 560
Max memory = 559154800
Elapsed time = 0.048 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 11593
Per table statistics:
------------------------------+---------+---------+---------+---------+---------+
 Table name                   | Natural | Index   | Insert  | Update  | Delete  |
------------------------------+---------+---------+---------+---------+---------+
SERVICE                       |         |    9582 |         |         |         |
------------------------------+---------+---------+---------+---------+---------+
```

12. UNEVEN DATA DISTRIBUTION IS IGNORED

```sql
SELECT
  COUNT(*)
FROM
  HORSE
  JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
  JOIN DEPARTURE ON DEPARTURE.CODE_DEPARTURE = HORSE.CODE_DEPARTURE
WHERE FARM.INN IS NULL
  AND HORSE.CODE_BREED = 65;
```



12. UNEVEN DATA DISTRIBUTION IS IGNORED

```
Current memory = 551859280
Delta memory = 384
Max memory = 552164464
Elapsed time = 24.407 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 1241189

Per table statistics:
------------------------------+---------+---------+---------+---------+---------+
 Table name                   | Natural | Index   | Insert  | Update  | Delete  |
------------------------------+---------+---------+---------+---------+---------+
COLOR                         |         |  116396 |         |         |         |
DEPARTURE                     |      19 |         |         |         |         |
FARM                          |         |   39254 |         |         |         |
HORSE                         |         |  116396 |         |         |         |
------------------------------+---------+---------+---------+---------+---------+
```

```
Select Expression
    -> Aggregate
        -> Filter
            -> Nested Loop Join (inner)
                -> Hash Join (inner)
                    -> Nested Loop Join (inner)
                        -> Filter
                            -> Table "FARM" Access By ID
                                -> Bitmap
                                    -> Index "IDX_FARM_INN" Range Scan (full match)
                        -> Filter
                            -> Table "HORSE" Access By ID
                                -> Bitmap And
                                    -> Bitmap
                                        -> Index "FK_HORSE_FARMBORN" Range Scan (full match)
                                    -> Bitmap
                                        -> Index "FK_HORSE_BREED" Range Scan (full match)
                    -> Record Buffer (record length: 25)
                        -> Table "DEPARTURE" Full Scan
                -> Filter
                    -> Table "COLOR" Access By ID
                        -> Bitmap
                            -> Index "PK_COLOR" Unique Scan
```

12. UNEVEN DATA DISTRIBUTION IS IGNORED

```
SELECT
  COUNT(*)
FROM
  HORSE
  JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
  JOIN DEPARTURE ON DEPARTURE.CODE_DEPARTURE =
HORSE.CODE_DEPARTURE
WHERE FARM.INN || '' IS NULL
  AND HORSE.CODE_BREED = 65;
```

12. UNEVEN DATA DISTRIBUTION IS IGNORED

```
Current memory = 552304224
Delta memory = 0
Max memory = 559519680
Elapsed time = 0.407 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 636023

Per table statistics:
------------------------------+---------+---------+---------+---------+---------+
 Table name                   | Natural | Index   | Insert  | Update  | Delete  |
------------------------------+---------+---------+---------+---------+---------+
COLOR                         |     242 |         |         |         |         |
DEPARTURE                     |      19 |         |         |         |         |
FARM                          |         |  122836 |         |         |         |
HORSE                         |         |  122836 |         |         |         |
------------------------------+---------+---------+---------+---------+---------+
```

```
Select Expression
    -> Aggregate
        -> Filter
            -> Nested Loop Join (inner)
                -> Hash Join (inner)
                    -> Nested Loop Join (inner)
                        -> Table "DEPARTURE" Full Scan
                        -> Filter
                            -> Table "HORSE" Access By ID
                                -> Bitmap And
                                    -> Bitmap
                                        -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)
                                    -> Bitmap
                                        -> Index "FK_HORSE_BREED" Range Scan (full match)
                    -> Record Buffer (record length: 25)
                        -> Table "COLOR" Full Scan
                -> Filter
                    -> Table "FARM" Access By ID
                        -> Bitmap
                            -> Index "PK_FARM" Unique Scan
```

12. UNEVEN DATA DISTRIBUTION IS IGNORED

```sql
SELECT
  COUNT(*)
FROM
  HORSE
  JOIN FARM ON FARM.CODE_FARM+0 = HORSE.CODE_FARM
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
  JOIN DEPARTURE ON DEPARTURE.CODE_DEPARTURE = HORSE.CODE_DEPARTURE
WHERE FARM.INN || '' IS NULL
  AND HORSE.CODE_BREED = 65;
```

12. UNEVEN DATA DISTRIBUTION IS IGNORED

```
Current memory = 552578032
Delta memory = 384
Max memory = 560648992
Elapsed time = 0.193 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 178133

Per table statistics:
------------------------------+---------+---------+---------+---------+---------+
 Table name                   | Natural | Index   | Insert  | Update  | Delete  |
------------------------------+---------+---------+---------+---------+---------+
COLOR                         |     242 |         |         |         |         |
DEPARTURE                     |      19 |         |         |         |         |
FARM                          |   41356 |         |         |         |         |
HORSE                         |         |  122836 |         |         |         |
------------------------------+---------+---------+---------+---------+---------+
```

```
Select Expression
    -> Aggregate
        -> Filter
            -> Hash Join (inner)
                -> Hash Join (inner)
                    -> Hash Join (inner)
                        -> Filter
                            -> Table "FARM" Full Scan
                        -> Record Buffer (record length: 49)
                            -> Filter
                                -> Table "HORSE" Access By ID
                                    -> Bitmap
                                        -> Index "FK_HORSE_BREED" Range Scan (full match)
                    -> Record Buffer (record length: 25)
                        -> Table "DEPARTURE" Full Scan
                -> Record Buffer (record length: 25)
                    -> Table "COLOR" Full Scan
```

12. UNEVEN DATA DISTRIBUTION - PARTIAL INDEX

```sql
-- CREATE INDEX IDX_FARM_INN ON FARM (INN);
DROP INDEX IDX_FARM_INN;
CREATE INDEX IDX_FARM_INN ON FARM (INN) WHERE (INN IS NOT NULL);

SELECT
  COUNT(*)
FROM
  HORSE
  JOIN FARM ON FARM.CODE_FARM+0 = HORSE.CODE_FARM
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
  JOIN DEPARTURE ON DEPARTURE.CODE_DEPARTURE = HORSE.CODE_DEPARTURE
WHERE FARM.INN IS NULL
  AND HORSE.CODE_BREED = 65;
```

```
Select Expression
    -> Aggregate
        -> Filter
            -> Hash Join (inner)
                -> Hash Join (inner)
                    -> Hash Join (inner)
                        -> Filter
                            -> Table "FARM" Full Scan
                        -> Record Buffer (record length: 49)
                            -> Filter
                                -> Table "HORSE" Access By ID
                                    -> Bitmap
                                        -> Index "FK_HORSE_BREED" Range Scan (full match)
                    -> Record Buffer (record length: 25)
                        -> Table "DEPARTURE" Full Scan
                -> Record Buffer (record length: 25)
                    -> Table "COLOR" Full Scan
```

Book “Secrets of Firebird Optimizer”

https://firebirdsql.org/en/featured-products/book-secrets-of-firebird-sql-optimizer

- Translations to 6 languages: English, Portuguese, Polish, German, and more!

Questions?

ak@firebirdsql.org