Under the Hood: Data Access Paths

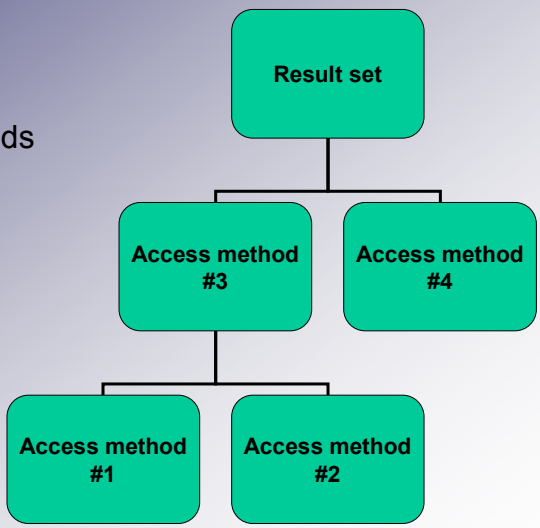
Arno Brinkman
&
Dimitry Yemanov

Intro

- **Data access paths, access methods and data streams**
- **Access methods by function:**
  - primary
  - filters
  - junctions
- **Primary methods create data streams, others just transform them**

Intro

- **Access path as a tree:**
  - root is a result set
  - nodes are access methods
  - links are data streams



Intro

- **Access methods by implementation:**
  - pipelined
  - buffered
- **Attributes of access methods:**
  - cardinality
  - cost
- **Logical reads (page fetches) as cost units**

Primary access methods (internal tables)

- **Direct table access**
  - Full table scan
  - Access via record identifier
  - Positioned access
- **Index based access**
  - Bitmaps
  - Range index scan
  - Index navigation

Full table scan

- **Pages are read sequentially in their storage order – highest throughput**
- **Rows are processed in the pipeline mode (one by one)**
- **Pages are not prefetched, rows are not buffered, no multi-block I/O**
- **Used only when there are no indices to be applied**
- **Reported as non indexed reads in statistics**

Full tables scan (example)

```
SELECT *
FROM RDB$RELATIONS

PLAN (RDB$RELATIONS NATURAL)

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


Access via record identifier

- A single row is read using its record number (aka DB_KEY value)
- Record number contains information about data page and offset (slot) on this page
- Used as implementation by other access methods, not exposed to the optimizer
- Reported as indexed reads in statistics

Positioned access

- Used for positioned forms of UPDATE and DELETE statements (syntax: WHERE CURRENT OF)
- It's not the same as access using RDB$DB_KEY
- Works with an already fetched row (active cursor), so no reads are actually performed

Index based access

- B+ tree index with prefix compression of keys
- Indices: single segment and composite
- Full and partial key matches
- No index-only scan due to MGA implementation
- Indices allows only uni-directional scan
- Selectivity of indices, assumption about uniform distribution of values

Bitmaps

- Effective solution to random data page I/O, no need in clustering indices or index-organized tables
- Implemented as a sparse bit array ordered by physical record numbers
- Bitmaps allow bitwise AND/OR operations to use any number of indices for a retrieval
- Uses access via record identifier to read the selected rows

Range index scan

- Index lookups, lower and upper bounds, setting the bitmap
- Full scan, range scan, equality scan, unique scan; special meaning of the unique scan
- Strict lower/upper bounds, solved in version 2.0
- Redundant scanning of NULL values, solved in version 2.0
- Cost based selection strategy

Range index scan (example)

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

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

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


Index navigation

- **Difference from a range scan - no bitmap**
- **Random data page I/O, to use with care**
- **Optimization of MIN/MAX**
- **Index based ordering (ORDER BY, GROUP BY, etc)**
- **Internal optimizations for search conditions:**
  - usage of lower/upper bounds to limit the scan
  - usage of bitmap to avoid unnecessary row reads

Index navigation (example)

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

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

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


Access using RDB$DB_KEY

- **Not officially documented, explained on www.cvalde.net**
- **Implemented via a singular bitmap**
- **Behaves like a pseudo index based access:**
  - INDEX clause in plans, but without an index name
  - reported as an indexed read

External tables

- Works similarly to a full table scan
- External file is read row by row, using the current pointer (offset) value
- No caching is done
- External tables cannot be indexed in the current implementation
- Cardinality is not calculated and assumed to be 10000

Procedures

- Used to retrieve rows from selectable procedures
- Procedure is always considered a black box, the engine makes no assumptions about its internals
- Any fetch from a procedure executes its code from the previous stall point to the next SUSPEND statement, then its output parameter values create a row
- Can be displayed differently in plans, depending on Firebird version

Filters

- These access methods transform the input data stream accordingly to their function
- Filter types by function:
  - Evaluation of booleans
  - External sorting
  - Aggregation
  - Counters
  - Singularity check
  - Record locking

Evaluation of booleans

- Most common type of a filter, used to implement clauses WHERE, HAVING, etc
- Check the given search condition for every input row and return only the satisfying rows
- Always performed as deep in the access path as possible, in order to reduce cardinality
- Booleans always duplicate index range scans to filter unnecessary rows out

Evaluation of booleans (example)

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

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

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


External sorting

- Used to order the input stream if navigation via an index cannot be applied
- Implements clauses ORDER BY, GROUP BY, DISTINCT etc, as well as creates the new index tree
- Requires to read all input rows before any of them goes to the output – buffered data source
- Quick sort + merge of runs, runs are stored in the external memory (VM or disk space)
- Normal and reducing modes

External sorting (example)

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

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

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


Aggregation

- Implement aggregate functions MIN/MAX/AVG/etc, including their usage in grouping
- Keeps a single register to calculate the function
- Requires an ordered input stream to perform a per group aggregation
- Intermediate reducing sorts are performed when AVG/SUM/COUNT are evaluated in the DISTINCT mode
- Hash aggregation as an alternative algorithm

Aggregation (example)

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

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

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


Counters

- **Most trivial type of filters**
- **Return only some part of the input stream, based on some value N of the internal counter**
- **Used to implement clauses FIRST/SKIP/ROWS**
- **Two kinds of this filter:**
  - FIRST counter (returns first N rows)
  - SKIP counter (return all rows starting with N+1)

Counters (example)

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

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

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


Singularity check

- Targeted to guarantee that the input stream contains only a single row
- Used to perform a runtime protection of singular subqueries
- Performs two fetches from the input stream, returns the first row if the second fetch encountered EOF, otherwise an error is thrown

Singularity check (example)

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

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

STATEMENT (SELECT)
[cardinality=1, cost=1.000]
  => SINGULAR
     [cardinality=1, cost=1.000]
    => TABLE (RDB$DATABASE) SEQUENTIAL ACCESS
       [cardinality=1, cost=1.000]
STATEMENT (SELECT)
[cardinality=1, cost=4.000]
  => BOOLEAN
     [cardinality=1, cost=4.000]
    => TABLE (RDB$RELATIONS) ACCESS BY DB_KEY
       [cardinality=1, cost=4.000]
      => BITMAP
         [cardinality=1, cost=3.000]
        => INDEX (RDB$INDEX_1) RANGE SCAN
           [cardinality=1, cost=3.000]
```


Record locking

- Implements pessimistic row level locking
- Used only if clause WITH LOCK is specified for a retrieval
- Creates a dummy record version marked by an identifier of the current transaction
- Allowed for primary access methods only

Record locking (example)

```
SELECT *
FROM RDB$RELATIONS
WITH LOCK

PLAN (RDB$RELATIONS NATURAL)

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


Junctions

- **These access methods are similar to filters, but deal with multiple input streams**
- **Junction types:**
  - Joins
    - **Nested iteration**
    - **One-way merge**
    - **Hash joins**
  - Unions

Joins

- Inner and outer joins; left, right and full outer joins
- Outer (master) and inner (slave) streams
- Join condition or lack or it (cross join)
- Independence of inner join streams and dependency of outer join streams, how the optimizer controls the order of streams

Nested iteration

- Also known as nested loops join or recursive join
- How it works: nested iterations on input streams
- High cost of duplicated inner reads, usage of an index scan to optimize the inner retrievals
- Used always when there are indices that could be applied for a join condition
- Complex AND/OR based join conditions are supported

Nested iteration (continued)

- Known pitfall: indices are never used for a full outer join
- How to determine extremely bad nested iterations looking at plans
- Where and why procedures are placed in the join order
- Known issue with procedures depending on other streams via input parameters, a workaround

Nested iteration (example)

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

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

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

One-way merge

- **An alternative algorithm to perform a join:**
  - input streams are independent,
    but required to be ordered by the join key
  - streams are merged on a per row basis
    by walking the binary merge tree
- **Allowed equality joins only, effectively performs expression based joins, outer joins cannot be handled**
- **Used when a recursive join cannot be applied effectively**

One-way merge (example)

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

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

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

Hash joins

- **Yet another alternative join algorithm**
- **Implementation details:**
  - input streams are always dependent
  - the inner (slave) stream is entirely read in advance, its join keys create a hash table
  - every row from the outer (master) stream is probed against the hash table
- **Same advantages and pitfalls as for a merge join**
- **Wins for small inner (slave) streams**

Unions

- **Responsible for the UNION operation**
- **Two modes:**
  - ALL (return everything)
  - DISTINCT (eliminate full duplicates of rows)
- **Sequentially return all rows from all input streams**
- **In the DISTINCT mode, also applies a reducing sorting filter to its output stream**

Unions (example)

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

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

STATEMENT (SELECT)
[cardinality=1500, cost=1500.000]
  => SORT
     [cardinality=1500, cost=1500.000]
    => UNION
       [cardinality=1500, cost=1500.000]
      => BOOLEAN
         [cardinality=500, cost=500.000]
        => TABLE (RDB$RELATIONS) SEQUENTIAL ACCESS
           [cardinality=500, cost=500.000]
      => BOOLEAN
         [cardinality=1000, cost=1000.000]
        => TABLE (RDB$PROCEDURES) SEQUENTIAL ACCESS
           [cardinality=1000, cost=1000.000]
```


The end

Thanks for your attention.

Speaker: **Arno Brinkman**
Prepared by: **Dimitry Yemanov**