Indexing & optimizations

Optimizations in v.2.1

Optimization improvements in v.2.1 include:

(v.2.1) Economising on indexed reads for MIN() and MAX()

Indexed MIN/MAX aggregates would produce three indexed reads instead of the expected single read. So, with an ASC index on the non-nullable COL, the query

```
SELECT MIN(COL) FROM TAB
```

should be completely equivalent, to

```
SELECT FIRST 1 COL FROM TAB
ORDER BY 1 ASC
```

with both performing a single record read. However, formerly, the first query required three indexed reads while the second one required just the expected single read. Now, they both resolve to a single read.

The same optimization applies to the MAX() function when mapped to a DESC index.

Improved PLAN clause

D. Yemanov

(**v.2.0.x**) A PLAN clause optionally allows you to provide your own instructions to the engine and have it ignore the plan supplied by the optimizer. Firebird 2 enhancements allow you to specify more possible paths for the engine. For example:

```
PLAN (A ORDER IDX1 INDEX (IDX2, IDX3))
```

For more details, please refer to the topic Query plans improvements in the DML chapter.

back to top of page

Optimizer improvements

This section represents a collection of changes done in Firebird 2 to optimize many aspects of performance.

For all databases

The first group of changes affect all databases, including those not yet upgraded to ODS 11.x.

Some general improvements

O. Loa, D. Yemanov

- Much faster algorithms to process the dirty pages tree.

Firebird 2 offers a more efficient processing of the list of modified pages, a.k.a. the dirty pages tree. It affects all kinds of batch data modifications performed in a single transaction and eliminates the known issues with performance getting slower when using a buffer cache of >10K pages.

This change also improves the overall performance of data modifications.

- Increased maximum page cache size to 128K pages (2GB for 16K page size)

Faster evaluation of IN() and OR

O. Loa

Constant IN predicate or multiple OR Booleans are now evaluated faster.

Sparse bitmap operations were optimized to handle multiple OR Booleans or an IN (<constant list>) predicate more efficiently, improving performance of these operations.

Improved UNIQUE retrieval

A. Brinkman

The optimizer will now use a more realistic cost value for unique retrieval.

More optimization of NOT conditions

D. Yemanov

NOT conditions are simplified and optimized via an index when possible.

Example

```
(NOT NOT A = 0) -> (A = 0)
(NOT A > 0) -> (A <= 0)
```

Distribute HAVING conjunctions to the WHERE clause

If a HAVING clause or any outer-level SELECT refers to a field being grouped by, this conjunct is distributed deeper in the execution path than the grouping, thus allowing an index scan to be used. In other words, it allows the HAVING clause not only be treated as the WHERE clause in this case, but also be optimized the same way.

Examples

```
select rdb$relation_id, count(*)
from rdb$relations
group by rdb$relation_id
having rdb$relation_id > 10

select * from (
  select rdb$relation_id, count(*)
  from rdb$relations
  group by rdb$relation_id
  ) as grp (id, cnt)
where grp.id > 10
```

In both cases, an index scan is performed instead of a full scan.

Distribute UNION conjunctions to the inner streams

Distribute UNION conjunctions to the inner streams when possible.

Improved handling of CROSS JOIN and Merge/SORT

Improved cross join and merge/sort handling.

Better choice of join order for mixed inner/outer joins

Reasonable join order for intermixed inner and outer joins.

Equality comparison on expressions

MERGE PLAN may now be generated for joins using equality comparison on expressions.

back to top of page

For ODS 11 databases only

This group of optimizations affects databases that were created or restored under Firebird 2 or higher.

Segment-level selectivities are used

See Selectivity maintenance per segment.

Better support for IS NULL and STARTING WITH

Previously, IS NULL and STARTING WITH predicates were optimized separately from others, thus causing non-optimal plans in complex ANDed/ORed Boolean expressions. From v2.0 and ODS 11, these predicates are optimized in a regular way and hence benefit from all possible optimization strategies.

Matching of both OR and AND nodes to indexes

Complex Boolean expressions consisting of many AND/OR predicates are now entirely mapped to available indices if at all possible. Previously, such complex expressions could be optimized badly.

Better JOIN orders

Cost estimations have been improved in order to improve JOIN orders.

Indexed order enabled for outer joins

It is now possible for indexed order to be utilised for outer joins, i.e. navigational walk.

back to top of page

Enhancements to indexing

252-byte index length limit is gone

A. Brinkman

New and reworked index code is very fast and tolerant of large numbers of duplicates. The old aggregate key length limit of 252 bytes is removed. Now the limit depends on page size: the maximum size of the key in bytes is 1/4 of the page size (512 on 2048, 1024 on 4096, etc.).

A 40-bit record number is included on “non leaf-level pages” and duplicates (key entries) are sorted by this number.

Expression indexes

O. Loa, D. Yemanov, A. Karyakin

Arbitrary expressions applied to values in a row in dynamic DDL can now be indexed, allowing indexed access paths to be available for search predicates that are based on expressions.

Syntax pattern

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]] INDEX <index name>
  ON <table name>
  COMPUTED BY ( <value expression> )
```

Examples

1.

```
CREATE INDEX IDX1 ON T1
   COMPUTED BY ( UPPER(COL1 COLLATE PXW_CYRL) );
COMMIT;
/**/
SELECT * FROM T1
   WHERE UPPER(COL1 COLLATE PXW_CYRL) = 'ÔÛÂÀ'
 -- PLAN (T1 INDEX (IDX1))
```

2.

```
CREATE INDEX IDX2 ON T2
   COMPUTED BY ( EXTRACT(YEAR FROM COL2) || EXTRACT(MONTH FROM COL2) );
COMMIT;
/**/
SELECT * FROM T2
   ORDER BY EXTRACT(YEAR FROM COL2) || EXTRACT(MONTH FROM COL2)
 -- PLAN (T2 ORDER IDX2)
```

Note:

- The expression used in the predicate must match exactly the expression used in the index declaration, in order to allow the engine to choose an indexed access path. The given index will not be available for any retrieval or sorting operation if the expressions do not match.
- Expression indices have exactly the same features and limitations as regular indices, except that, by definition, they cannot be composite (multi-segment).

back to top of page

Changes to NULL keys handling

V. Horsun, A. Brinkman

- Null keys are now bypassed for uniqueness checks. (V. Horsun)

If a new key is inserted into a unique index, the engine skips all NULL keys before starting to check for key duplication. It means a performance benefit as, from v.1.5 on, NULLs have not been considered as duplicates.

- NULLs are ignored during the index scan, when it makes sense to ignore them. (A. Brinkman).

Prevously, NULL keys were always scanned for all predicates. Starting with v.2.0, NULL keys are usually skipped before the scan begins, thus allowing faster index scans.

Note: The predicates IS NULL and IS NOT DISTINCT FROM still require scanning of NULL keys and they disable the aforementioned optimization.

Improved index compression

A. Brinkman

A full reworking of the index compression algorithm has made a manifold improvement in the performance of many queries.

Selectivity maintenance per segment

D. Yemanov, A. Brinkman

Index selectivities are now stored on a per-segment basis. This means that, for a compound index on columns (A, B, C), three selectivity values will be calculated, reflecting a full index match as well as all partial matches. That is to say, the selectivity of the multi-segment index involves those of segment A alone (as it would be if it were a single-segment index), segments A and B combined (as it would be if it were a double-segment index) and the full three-segment match (A, B, C), i.e., all the ways a compound index can be used.

This opens more opportunities to the optimizer for clever access path decisions in cases involving partial index matches.

The per-segment selectivity values are stored in the column RDB$STATISTICS of table RDB$INDEX_SEGMENTS. The column of the same name in RDB$INDICES is kept for compatibility and still represents the total index selectivity, that is used for a full index match.