Firebird Conference 2019
Berlin, 17-19 October

IBSurgeon

THE WHOLE TRUTH ABOUT JOINS

What kinds of JOINs exist

- INNER and OUTER
- Outer: LEFT, RIGHT, FULL OUTER
  - RIGHT → LEFT
  - FULL OUTER — somewhat exotic (with optimization caveats)

Differences in optimization

- INNER: join order does not matter
  - But directly affects performance
  - Optimizer uses statistics and sometimes heuristics
- OUTER: join order is pre-defined by the SQL statement
  - No ways for permutations

Differences in optimization

- INNER: all predicates are equivalent (ON = WHERE)
  - Combined together by the optimizer
  - Dependent predicates are used for joining
  - Independent predicates are used for filtering

Differences in optimization

- OUTER: there is a difference between ON and WHERE
  - But sometimes predicates can «migrate» from WHERE to ON

```
TABLE_A TA
left join TABLE_B TB on TA.ID = TB.ID
where TB.STATUS = 1

vs
where TB.STATUS is null

vs
where TB.STATUS is not null
```

Differences in optimization

- INNER: all streams are optimized together
  - A join B join C join D — single 4-way join
    `JOIN (A, B, C, D)`
  - Order of tables is determined by the optimizer
- OUTER: tables are combined into pairs
  - A left join B left join C — two joins
    `((A JOIN B) JOIN C)`

Differences in optimization

- Mixing INNER/OUTER separates tables into groups
  - ```
    A join B on A.F1 = B.F1
    left join C on B.F2 = C.F2
    join D on C.F3 = D.F3
    ```
  - Three joins
    ```
    (((A JOIN B) JOIN C) JOIN D)
    ```
  - Permutations are possible between {A, B},
    D is optimized independently
    ```

Differences in optimization

- Mixing INNER/OUTER separates tables into groups
  - ```
    A join B on A.F1 = B.F1
    left join C on B.F2 = C.F2
    join D on C.F3 = D.F3
    ```
  - Three joins
    ```
    (((A JOIN B) JOIN C) JOIN D)
    ```
  - Permutations are possible between {A, B},
    D is optimized independently
- Hint!
  - All INNER streams should go first
    and only then OUTER streams should follow
    ```

Differences in optimization

- INNER: views / derived tables can be merged
  - `V = (B join C)`
    `A join V`
    `=`
    `A join B join C`
  - Then they are optimized as a simple 3-way join
- OUTER: such a merging is impossible
  - We will see groups of multiple joins

Hinting

- Via enable/disable index usage
  - + 0 for numbers / dates, || `` for strings
  - Indirectly affects join order
- Via LEFT/RIGHT JOIN
  - Affects only join order

```
select *
from TABLE_A TA
left join TABLE_B TB on TA.ID = TB.ID
where TB.ID is not null
```

JOIN execution algorithms

- Nested loop join
  - JOIN in query plan
- Merge join (aka sort/merge join)
  - MERGE in query plan
- Hash join – starting with Firebird 3
  - HASH in query plan

Nested loop join

- ```
  A join B join C
  =
  for select from A
    for select from B
      for select from C
  ```

- Without join conditions — decart product (aka CROSS JOIN), very slow (nested full scans)
- Independent predicates allow to use indices for filtering and thus limit record sets
- Dependent predicates (join conditions) allow to execute context-based retrieval
  ```

Nested loop join

- Optimizer goal — reduce record sets for nested streams by using properly indexed retrievals

- How cost is calculated:

```
A join B
```

cost(A) + cardinality(A) * cost(B)

```
A join B join C
```

cost(A) + cardinality(A) * cost(B) +
cardinality(A, B) * cost(C)

Merge join

- All input streams are SORTed
- One-way merge is performed
- Indices are used for filtering only

- Sorting costs a lot, swapping to temp files is possible
- Now it works for equi-joins only
- Now used for INNER JOINs only
- Temporarily disabled in Firebird 3

Hash join

- Smaller stream is buffered inside the temp space, hash table is built for all join keys
- Larger table is scanned once, join keys are probed against the hash table
- Indices are used for filtering only

- Hashing is not free either
- Possible for equi-joins only
- Now used for INNER JOINs only
- Firebird 3 temporarily uses HASH JOIN intead of MERGE JOIN

When MERGE is better than HASH

- At least one input stream is already sorted by the join key
- ORDER BY the join key exists
- Joined streams all very large

- There is no way to choose between them now :-(

When HASH is better than Nested Loops

- Many retrievals from the nested streams
  - Outer stream cardinality
    vs
    Inner stream selectivity
  - Complex computations inside the inner streams

- Can be «hinted» by disabling indices on both join fields

How join algorithm is chosen now

- If there are indexed join conditions
  → nested loop join
- If there are no indexed join conditions
  AND they are equalities
  AND it is INNER JOIN
  → merge join / hash join
- If join conditions are inequalities
  OR it is OUTER JOIN
  → nested loop join

What can be changed in the near future

- Merge/Hash join implementation for OUTER JOINs
- Cost- (or heuristic-) based choice between merge and hash joins
- Cost-based choice between nested loop and merge/hash algorithms

Joins with selectable stored procedures

- Optimizer puts SP at the first position
  - To avoid multiple executions of SP
  - To use indices for joined table(s)
  - And that is good :-)
  - If index is «turned off» via a hint, MERGE/HASH will be used instead of Nested Loops, but usually it does not make much sense

Joins with selectable stored procedures

- Join via input parameter
  - Before Firebird 3 — error «no record to fetch», LEFT JOIN should be used instead of INNER JOIN
  - Now optimizer puts the procedure at its proper position

```
TABLE_A TA
join PROC_B(TA.ID) on 1 = 1
```

or

```
TABLE_A TA cross join PROC_B(TA.ID)
```

Joins with aggregates / unions

- They are also positioned unconditionally
- But it is not always good
  (if predicate pushing is possible)
- Can be altered via LEFT JOIN

```
select …
from TABLE_A TA
  join ( select FLD1, sum(FLD2)
           from …
           group by FLD1 ) DT
    on TA.ID = DT.FLD1
```

Joins and ORDER BY

- Ordering via SORT allows any possible join order, let the optimizer doing its work
- ORDER plan (index-order navigation) can happen only for the first joined table
- SORT vs ORDER — to be explained
- Join order can be altered by «hinting», but is it really necessary?
- Heuristics for FIRST, MIN/MAX, EXISTS

Example (nested loop join)

```
select s_acctbal, s_name, n_name
from part
  join partsupp on p_partkey = ps_partkey
  join supplier on ps_suppkey = s_suppkey
  join nation on s_nationkey = n_nationkey
  join region on n_regionkey = r_regionkey
where p_size = 15
  and p_type like '%BRASS'
  and r_name = 'EUROPE'
order by
  s_acctbal, n_name, s_name, p_partkey
```

Example (nested loop join)

```
PLAN SORT (JOIN (
  NATION NATURAL,
  REGION INDEX (REGION_PK),
  SUPPLIER INDEX (SUPPLIER_NATIONKEY),
  PARTSUPP INDEX (PARTSUPP_SUPPKEY),
  PART INDEX (PART_PK)))
```

vs

```
PLAN SORT (JOIN (
  PART NATURAL,
  PARTSUPP INDEX (PARTSUPP_PK),
  SUPPLIER INDEX (SUPPLIER_PK),
  NATION INDEX (NATION_PK),
  REGION INDEX (REGION_PK)))
```

Example (nested loop join)

```
→ Sort
  → Nested Loop Join (inner)
    → Table «NATION» Full Scan
    → Table «REGION» Access By ID
      → Bitmap
        → Index «REGION_PK» Unique Scan
    → Table «SUPPLIER» Access By ID
      → Bitmap
        → Index «SUPPLIER_NATIONKEY» Range Scan
    → Table «PARTSUPP» Access By ID
      → Bitmap
        → Index «PARTSUPP_SUPPKEY» Range Scan
    → Table «PART» Access By ID
      → Bitmap
        → Index «PART_PK» Unique Scan
```

Example (nested loop join)

```
→ Sort
  → Nested Loop Join (inner)
    → Table «PART» Full Scan
    → Table «PARTSUPP» Access By ID
      → Bitmap
        → Index «PARTSUPP_PK» Unique Scan
    → Table «SUPPLIER» Access By ID
      → Bitmap
        → Index «SUPPLIER_PK» Unique Scan
    → Table «NATION» Access By ID
      → Bitmap
        → Index «NATION_PK» Unique Scan
    → Table «REGION» Access By ID
      → Bitmap
        → Index «REGION_PK» Unique Scan
```

Example (nested loop join)

```
select s_acctbal, s_name, n_name
from part
  join partsupp on p_partkey = ps_partkey
  join supplier on ps_suppkey = s_suppkey
  join nation on s_nationkey+0 = n_nationkey+0
  join region on n_regionkey+0 = r_regionkey+0
where p_size = 15
  and p_type like '%BRASS'
  and r_name = 'EUROPE'
order by
  s_acctbal, n_name, s_name, p_partkey
```

Example (nested loop join)

```
PLAN SORT (
  HASH (
    HASH (
      JOIN (
        PART NATURAL,
        PARTSUPP INDEX (PARTSUPP_PK),
        SUPPLIER INDEX (SUPPLIER_PK)
      ),
      NATION NATURAL
    ),
    REGION NATURAL)
  )
)
```

Example (nested loop join)

```
→ Sort
  → Hash Join (inner)
    → Hash Join (inner)
      → Nested Loop Join (inner)
        → Table «PART» Full Scan
        → Table «PARTSUPP» Access By ID
          → Bitmap
            → Index «PARTSUPP_PK» Unique Scan
        → Table «SUPPLIER» Access By ID
          → Bitmap
            → Index «SUPPLIER_PK» Unique Scan
      → Table «NATION» Full Scan
    → Table «REGION» Full Scan
```

Example (nested loop join)

```
select l_orderkey, o_orderdate,
  o_shippriority, sum(l_extendedprice)
from customer
  join orders on c_custkey = o_custkey
  join lineitem on o_orderkey = l_orderkey
where c_mktsegment = 'BUILDING'
  and o_orderdate < date '1995-03-15'
  and l_shipdate > date '1995-03-15'
group by 1, 2, 3
```

Example (nested loop join)

```
PLAN SORT (JOIN (
   CUSTOMER NATURAL,
   ORDERS INDEX (ORDERS_CUSTKEY),
   LINEITEM INDEX (LINEITEM_PK, LINEITEM_SHIPDATE)))
```

vs

```
PLAN SORT (JOIN (
   LINEITEM INDEX (LINEITEM_SHIPDATE),
   ORDERS INDEX (ORDERS_PK),
   CUSTOMER INDEX (CUSTOMER_PK)))
```

Example (nested loop join)

```
PLAN SORT (JOIN (
  LINEITEM INDEX (LINEITEM_SHIPDATE),
  ORDERS INDEX (ORDERS_PK),
  CUSTOMER INDEX (CUSTOMER_PK)))
```

vs

```
PLAN SORT (JOIN (
  ORDERS INDEX (ORDERS_ORDERDATE),
  CUSTOMER INDEX (CUSTOMER_PK),
  LINEITEM INDEX (LINEITEM_PK)))
```