Optimizer: Issues and Solutions

Dmitry Yemanov
mailto:dimitr@firebirdsql.org

Firebird Project
http://www.firebirdsql.org/

History

- **Legacy optimizer**
  - Cost-based calculations for inner joins
  - Heuristics for everything else
  - Totally wrong for partial matches and non-equality comparisons
  - Ineffective for complex joins
  - Often unable to use an index in sub-queries

Optimizer "v2": Firebird 2 and ODS 11

- **New statistics**
  - Per-segment index selectivity
- **New features**
  - Exact cardinality calculation for small tables
  - Actual selectivity depends on the predicate
  - Brand new cost calculation for retrievals
  - More efficient cost calculation for chained joins
  - Predicate "pushing" for aggregates, unions and derived tables

Remaining issues

- **Insufficient statistics**
  - For big tables, cardinality is estimated based on data page count and record compression ratio
  - Index tree depth and index key compression ratio are hardcoded
  - No difference between selectivities for NULLs and non-NULLs
  - Uniform value distribution is supposed
  - Selectivities for non-indexed columns are lacking

Remaining issues

- **Procedures, aggregates and unions in joins**
  - Lack of cost estimations
  - Predefined join order
  - Problems with correlated procedures
  - Other complex derived tables and CTEs can be also affected

Remaining issues

- **Example 1**

```
select e.emp_no, e.salary
from employee e
join ( select first 10
         ex.dept_no, sum(ex.salary)
       from employee ex
       group by 1
       order by 1 desc ) ea
  on e.dept_no = ea.dept_no
```

- **Example 2**

```
select *
from table1 t1
  cross join proc1(t1.id)
```

Remaining issues

- **Chained joins**
  - Hard to estimate the cost properly
  - “Hints” could be used to vary the join order
  - No cost based choice between nested loop join and merge join

Remaining issues

- **Example**

```
select *
from customer, orders, lineitem,
  supplier, nation, region
where
  c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and l_suppkey = s_suppkey
  and c_nationkey = s_nationkey
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_name = 'ASIA'
  and o_orderdate >= date '1994-01-01'
  and o_orderdate < date '1995-01-01'
```

Remaining issues

- **Outer joins**
  - No cost estimations at all
  - Single execution path: nested loop join
- **Example**

```
select *
from employee e
left join department d
  on e.dept_no = some_func(d.dept_no)
```

Remaining issues

- **Sub-queries**
  - Uncorrelated (invariant)
  - Non-indexed correlations
  - Implicitly correlated
- **IN predicate**
  - The correlation is injected by the engine
  - Consider IN vs EXISTS
  - NOT IN <> NOT EXISTS !!!

Remaining issues

- **Example 1**

```
select e.emp_no, e.salary
from employee e
left join ( select avg(ex.salary) avgval
              from employee ex ) ea
  on e.salary > ea.avgval
```

- **Example 2**

```
select *
from employee e
where e.dept in
  ( select d.dept_id
    from departments d
    where d.flag = 'X' )
```

Remaining issues

- **Performance of sorting**
  - Sort record: sort key, dbkey, txn id, other fields
  - Favoring sequential reads over random reads
  - Fixed size, hence unpacked fields
  - Sorting levels: internal buffer → temp cache → disk
  - Wider record means more I/O in the temp space

Remaining issues

- **Slow example**

```
select t1.int_field, t1.varchar300_field
from table1 t1
order by t1.int_field
```

- **Fast example**

```
select t1.int_field, t2.varchar300_field
from ( select t1.int_field, t1.id
       from table1 t1
       order by t1.int_field ) t1
join table1 t2 on t1.id+0 = t2.id
```

Optimizer “v3”: Firebird 3 and ODS 12

- **New statistics**
  - Table: number of pages, number of rows
  - Index: depth, number of leaf pages, number of nodes, clustering factor
  - Column: number of NULLs, selectivity for non-NULL values
  - Value distribution histograms
  - Complete or sampled

Optimizer “v3”: Firebird 3 and ODS 12

- **Alternative approach to external sorting**
  - Read and process only sort keys and dbkeys
  - While fetching, read the rows again via dbkey
  - Take the necessary fields from there
  - Consider the extra costs
  - Decide based on the available statistics

Optimizer "v3": Firebird 3 and ODS 12

- **Materialized sub-queries**
  - Underlying stream is sequentially read and cached inside a record buffer
  - Both sequential and random access are supported
  - Storage is provided by the temporary space manager and is dynamically balanced between memory and disk
  - CPU vs storage I/O

Optimizer "v3": Firebird 3 and ODS 12

- **Hash joins**
  - Larger input stream becomes an outer one, smaller input stream becomes an inner one
  - Inner stream is read in advance row by row, hash value is calculated for the join keys, row is stored in the buffer corresponding to the given hash group
  - Outer stream is read sequentially, hash value is calculated for the join keys and probed against the hash table
  - For a positive match, collisions are compared by the binary comparison of the join keys

Optimizer “v3”: Firebird 3 and ODS 12

- **Hash aggregation**
  - Every distinct grouping key has a corresponding entry in the hash table
  - Every aggregate function has its counter there
  - Backing sort is not needed
  - Efficient for non-selective grouping keys
  - Not much suitable for the FIRST <n> clause

Optimizer “v3”: Firebird 3 and ODS 12

- **Optimizer hints**
  - OPTIMIZE FOR { ALL | FIRST } ROWS
  - ORDER plan is preferred over SORT as much a possible, including inverted join orders
  - Nested loop joins are preferred to hash joins and especially merge joins
  - SORT plan is executed using the “sort faster, fetch slower” approach
  - Other optimizations are possible

Optimizer "v3": Firebird 3 and ODS 12

- **Optimizer improvements**
  - Estimate cardinality and cost through the whole data access path
  - Choose between nested loop joins and hash/merge joins based on cost
  - Materialize invariant sub-queries and non-indexed slave streams
  - Consider hash/merge joins for outer joins
  - Detect constant "always true" or "always false" predicates and skip unnecessary retrievals

Questions?