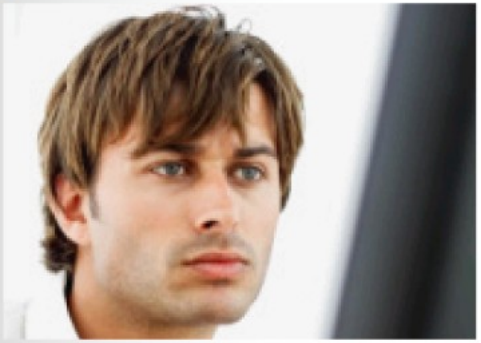
INGRES



Challenges in Query Optimization

Doug Inkster, Ingres Corp.

Abstract

- Some queries are inherently more difficult than others for a query optimizer to generate efficient plans. This session discusses the major phases of query compilation and sample optimizations that Ingres introduces at each phase. It then itemizes classes of difficult queries and describes recent changes to Ingres and changes under active consideration to assure the selection of the best possible plans

Overview

- **Phases of query compilation**
  - Parsing
  - Query rewrite
  - Enumeration/cost estimation
  - Code generation
- **Recent enhancements**
- **Difficult queries to optimize**

Parsing

- Transforms syntax into internal (usually tree) form for subsequent compilation
- No optimization but may introduce some transformations
- “where a between 10 and 20” becomes “where a >= 10 and a <= 20”
- “where x in (2, 5, 10)” used to become “where x = 2 or x = 5 or x = 10”, but no more (large lists caused stack overflows in server)

Query Rewrite

- **Query transformed mechanically to “canonical” form to increase optimization potential**
  - Views flattened into query
  - Subselects flattened into joins with containing query
  - DeMorgan's laws (“not (a < b)” becomes “a >= b”, etc.)
- **Query broken into “blocks”, each individually optimized then tied together in final query plan**
  - Multiple selects in union query
  - Join of aggregate view to other tables/views
- **Predicate movearound**
  - Push predicates from main query close to base tables

Enumeration/Cost Estimation

- **Enumeration generates all possible plans to evaluate a query**
  - Different join tree shapes
  - Different table orders
  - Different combinations of secondary indexes with base tables
  - Search space trimmed by heuristics
- **Cost model attaches cost estimates (CPU, disk I/O) to each potential plan**
  - Different formulas for each table access type, join type
  - Sort cost estimates, as necessary
- **Least expensive estimate is chosen**

Code Generation

- **Proposed plan is turned into code form executed by Ingres**
- **Nodes represent table access, join types, sorts**
- **Expression code generated for:**
  - Predicate evaluation
  - Arithmetic computations
  - Projections
- **Optimizations for keyed retrievals**
- **Optimizations to compiled expressions**
  - Minimize data movement, optimize path through ANDs, ORs

Recent Enhancements - Parser

- **avg() transformed to sum()/count()**
- **Addition of SYMMETRIC/ASYMMETRIC options to BETWEEN predicate**
  - “where a between symmetric x and y” becomes “where (a >= x and a <= y) or (a <= x and a >= y)”

Recent Enhancements – Query Rewrite

- **More predicate movearound – inspired by TPC H q20**
  - ... where ps_partkey in (select p_partkey from part where p_name like 'forest%')
    and ps_availqty > (select 0.5 * sum(l_quantity) from lineitem
    where l_partkey = ps_partkey and l_suppkey = ps_suppkey and l_shipdate >=
    '1994-01-01')
- **“ps_partkey in (select p_partkey ...” and “ps_partkey = l_partkey” imply that lineitem subselect can be transformed to:**
  “... from lineitem, part where l_partkey = ps_partkey and p_partkey = l_partkey and p_name like 'forest%'”
- **More restrictive lineitem subselect executes far more efficiently (40 seconds down to 2 seconds)**

Recent Enhancements – Enumeration/Cost Estimation

- **Greedy enumeration heuristic makes enumeration of large queries run waaaay faster**
- **optimizedb enhancements improve selectivity, cardinality estimates**
  - Up to 32000 cells
  - More exact cells in inexact histograms
  - Changed default histogram sizes
- **Updated sort cost estimation**
  - Tests showed sort CPU estimates to be 1000 times too large

Recent Enhancements – Code Generation

- **Common subexpression factoring inspired by TPC H q1**
  - select l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price, sum(l_extendedprice * (1 - l_discount)) as sum_disc_price, sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge, avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc, count(*) as count_order from ...
- **Locates and pre-evaluates common subexpressions into temporary variables**
  - (1 – l_discount), (l_extendedprice * (1 - l_discount))
- **Computes avg() as sum()/count(), converts count(expr) to count(*) when expr is not nullable**
  - count(*) computed once
  - sum(l_extendedprice) computed once, used in sum & avg

Optimizer Enhancements

- **Changes to parser, query rewrite improve optimization potential**
- **Changes to enumeration increase potential to consider best plans**
- **Changes to cost estimation improve accuracy and likelihood of identifying best plan**
  - Predicate selectivity, intermediate result cardinality are most important estimates to get right
- **Changes to code generation improve quality and efficiency of expression evaluation**

Problem Queries – Host Language Parameters

- Ingres splices parameter values into syntax for non-repeat queries
- Repeat queries are compiled with initial parameter values
- “... where col_a > :a_value” might qualify 5 rows or 500000 rows
- Best strategy might be an index scan or a full table scan – no way to know when query is compiled
- Poor row estimates feed into joins and result in even poorer estimates

Possible Solutions – Host Language Parameters

- **Multi-strategy query plans**
  - Query plan has a “fork” node above 2 access nodes for same table (one index, one table scan)
  - Test is executed in fork to determine which subnode to read rows from
  - Based on comparison of host parameter value with pre-determined (by optimizer from histogram) constant
- **Works for queries with few tables, could even include different join strategies in fork**
- **Too complex with large numbers of joins**

Problem Queries – Predicates with Functions/Expressions

- Selectivity of predicates with functions/expressions involving columns is very difficult to estimate
- Ingres doesn't even evaluate constant functions before estimating selectivities (e.g. date('1998-01-30'))

Possible Solutions - Predicates with Functions/Expressions

- Pre-evaluation of constant functions (typically cast functions) is long overdue in Ingres
- ANSI date/time support includes native literals allowing date/time values with no function notation e.g. date'1998-01-30'
- A more ambitious enhancement addressing single column functions/expressions would be to execute the function/expression on column histogram values to generate new (temporary) histogram

Problem Queries – Like Predicates

- **Like patterns starting with non-wild card generate encompassing BETWEEN predicate that optimizer can use to estimate selectivities**
  - "… where name like 'm%'" can be augmented by "… name >= 'm' and name < 'n'"
- **Like patterns starting with wild card cannot be used to estimate selectivity**
  - "… where name like '__m%'" is of no help

Possible Solutions – Like Predicates

- Like predicate could be executed on histogram values
- This would work best with accurate histograms (many exact cells)

Problem Queries – Multi-Column Restrictions

- **... where a > 25 and b = 10 and c < 19 ...**
- **Optimizers assume column value independence**
- **If a > 25 qualifies 10% of table, b = 10 qualifies 2%, c < 19 qualifies 20%, optimizer computes .1 * .02 * .2 = .04%**
- **Independence is rarely the right choice – there is usually correlation amongst columns**
- **Consider “city = 'burlington'” v.s. “city = 'burlington' and state = 'ma'”, then “city = 'worcester'” v.s. “city = 'worcester' and state = 'ma'”**
  - Indepence not bad for 1st case (lots of states with Burlington), but not good for 2nd case – result may be poorly chosen index scans, key joins

Possible Solutions – Multi-Attribute Restrictions

- **Trace point op189 dampens independence, but formula is not based in actual data**
- **Composite histograms on multi-column indexes**
  - Better than independence assumption
  - Need improved processing (Ingres 2006 doesn't fully exploit)
  - Use of combined column cardinality v.s. individual column cardinalities (a strong correlation measure)
- **Other multi-attribute histograms structures have been proposed, but offer no more than Ingres composite histograms**
- **Extend composite histograms to non-indexed columns**

Problem Queries - Joins

- **Join estimation is a different problem from restrictions**
  - Latter can produce precise estimates with histograms and constant comparisons
  - Former relies (in Ingres) on intersecting histogram cells of join columns
  - Histograms are required for both columns for accuracy
- **Containment principle used by most optimizers (not Ingres) says all rows of smaller table will join**
- **Even so, tendency is to underestimate join cardinality, especially for multi-attribute joins (column independence again)**

Possible Solutions - Joins

- **With accurate histograms Ingres does good job**
- **Possible new catalog to allow optimizer to identify joins that map onto referential relationships (join estimates become trivial)**
- **Composite histograms for multi-attribute joins**
- **Possible new column comparison catalog**
  - Proportion of rows in cross product for which join predicate is <, =, >
  - For single and multi-attribute joins
- **Restrictions on join columns also require histograms**

Problem Queries – Aggregate Joined to Non-Aggregate

- **Aggregate view or derived table joined to something else**
- **Estimation of aggregate result cardinality is difficult**
  - Product of grouping column cardinalities
  - Correlation reduces number of distinct groups
- **Joining on grouping columns is difficult, on any other result column is impossible to estimate**

Possible Solutions – Aggregate Joined to Non-Aggregate

- **Histogram can be constructed for result (one row per distinct set of grouping columns)**
- **Join can be estimated from histograms**
  - Histograms required on grouping columns, join columns

Problem Queries – No Current Solution

- **Restricting join on non-join columns**
  - No way to know value distribution of restriction column in join result, so no way to estimate selectivity of predicate
- **Subselect joins**
  - Usually flattened to equijoins
  - When not flattened (as with certain difficult cases) they are too complex to make informed cardinality estimates

Hints

- **Ultimately, hints allow user to control optimization of problem queries**
- **Prototyped in 2006u1, likely to appear in 2007**
  - Identify query result cardinality (possibly even at individual table level) to solve host parameter and restriction selectivity problems
  - Join order specification to solve some join problems
  - Explicit index specification to handle tables with multiple indexes and correlated restriction problems
  - Explicit join specification (request specific join technique between 2 tables) to solve join problems

Summary

- Query optimization is largely about estimation of predicate selectivity and result cardinalities
- Improving and fine tuning query optimization is a never ending task
- There are always classes of queries that defy optimization
- Numerous enhancements are currently planned to address optimization issues
- Hints

Questions