

# 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 stage. 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
  - Plan enumeration/cost model
  - Code generation
- Recent enhancements
- Difficult queries to optimize

# Parsing

- Transforms syntax into internal (usually tree) form for subsequent compilation
- No optimizations but may introduce some transformations
- “where a between 10 and 20” becomes “where a  $\geq$  10 and a  $\leq$  20”
- “where x in (2, 5, 10)” used to become “where a = 2 or a = 5 or a = 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 queries
  - 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 table access

## Enumeration/Cost Estimation

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

## Code Generation

- Proposed plan is turned into code form executed by Ingres
- Query plan nodes represent table access, join types, sorts
- Expression code generated for:
  - Predicate evaluation (where clause)
  - Arithmetic computations
  - Projections (select-list)
- Optimizations for keyed retrievals
- Optimizations to compiled expressions
  - Minimize data movement, optimize execution path through ANDs, ORs

## Recent Enhancements - Parser

- avg() transformed to sum()/count()
- Addition of SYMMETRIC/ASSYMETRIC options to BETWEEN predicate
  - “where a between symmetric x and y” becomes “where a  $\geq$  x and a  $\leq$  y or a  $\geq$  y and a  $\leq$  x”
- New syntax can make queries easier to express and more optimizeable

## 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 l\_partkey = p\_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 waaaaay faster
- optimizedb enhancements improve selectivity, cardinality estimates
  - Up to 32000 cells
  - More exact cells in inexact histograms
  - Increased 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 the 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 at compile time
- Poor row estimates feed into joins and cause even poorer estimates (and plans)

## Possible Solutions – Host Language Parameters

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

## Problem Queries – Multi-column Restrictions

- ... where  $a > 19$  and  $b = 10$  and  $c < 25$  ...
- All optimizers assume column value independence
- If  $a > 19$  qualifies 10% of rows,  $b = 10$  qualifies 2% and  $c < 25$  qualifies 20%, optimizer computes  $.1 * .02 * .2 = 0.04\%$
- Independence is rarely the right choice – there is usually some correlation amongst columns
- Consider “city = ‘burlington’” v.s. “city = ‘burlington’ and state = ‘ma’”, then “city = ‘worchester’” v.s. “city = ‘worchester’ and state = ‘ma’”
  - Independence is not a bad assumption for case 1 (there are lots of states with Burlington), but not good for case 2
  - Result may be poorly chosen index scans or key joins

# Possible Solutions – Multi-attribute Restrictions

- Trace point op189 dampens independence assumption, 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)
  - Comparison of combined column cardinality v.s. individual column cardinalities is a strong correlation measure
- Other multi-attribute histogram 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
  - Restrictions can produce precise estimates from histograms and constant comparisons
  - Joins rely (in Ingres) on intersection of histogram cells of join columns
  - Histograms are required on both columns for accuracy
- Containment principle used by most optimizers (but not Ingres) says all qualified rows of smaller table will enter 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 a good job
- Possible new catalog to allow optimizer to identify joins that map onto referential relationships (join estimates become trivial as all referencing rows have a match)
- 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 – No Current Solution

- Restricting join on non-join columns
  - E.g. `select * from cust, order where cust.cno = order.cno and order.channel = 'web'`
  - No way to know 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 2006r1, likely to appear in 2007
  - Identify query result cardinality (possibly even at individual table level) to solve host parameter and selectivity restriction problems
  - Join order specification to solve some join problems
  - Explicit index specification to handle tables with multiple secondary 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
  - Every Ingres release has incorporated optimization improvements
- There are always classes of queries that defy effective query optimization (in all DBMS')
- Numerous enhancements are currently planned to address optimization issues in Ingres
- Hints