CS262, Spring 2007 (Hellerstein/Brewer): SQL Query Optimization

- Basics
  - Given: A query joining n tables
  - The "Plan Space": Huge number of alternative, semantically equivalent plans.
  - The Perils of Error: Running time of plans can vary by many orders of magnitude
  - Ideal Goal: Map a declarative query to the most efficient plan tree.
  - Conventional Wisdom: You're OK if you avoid the rotten plans.
  - Industrial State of the art: Most optimizers use System R technique and work "OK" up to about 10 joins.
  - Wide variability in handling complex queries with aggregation, subqueries, etc.

- Approach 1: The Optimization Oracle
  - (definitely not to be confused with the company of the same name)
  - You'd like to get the following information, but in 0 time:
    - Consider each possible plan in turn.
    - Run it & measure performance.
    - The one that was fastest is the keeper.

- OK,OK. Approach 3: Think!
  - Three issues:
    - define plan space to search
    - do cost estimation for plans
    - find an efficient algorithm to search through plan space for "cheapest" plan
  - Selinger & the System R crowd the first to do this right. The Bible of Query Optimization.

- SQL Refresher
  - SELECT {DISTINCT} <list of columns>
  - FROM <list of tables>
  - {WHERE <list of "Boolean Factors" (predicates in CNF)>}
  - {GROUP BY <list of columns>}
  - {HAVING <list of Boolean Factors>}}
  - {ORDER BY <list of columns>};
  - Semantics are:
    - take Cartesian product (a/k/a cross-product) of tables in FROM clause, projecting to only those columns that appear in other clauses
    - if there's a WHERE clause, apply all filters in it
    - if there's a GROUP BY clause, form groups on the result
    - if there's a HAVING clause, filter groups with it
    - if there's an ORDER BY clause, make sure output is in the right order
    - if there's a DISTINCT modifier, remove dupes
    - Of course the plans don't do this exactly; query optimization interleaves 1 & 2 into a plan tree. GROUP BY, HAVING, DISTINCT and ORDER BY are applied at the end, pretty much in that order.

- Plan Space
  - All your favorite query processing algorithms:
CS262, Spring 2007 (Hellerstein/Brewer) : SQL Query Optimization

- sequential & index (clustered/unclustered) scans
- NL-join, (sort)-merge join, hash join
- sorting & hash-based grouping
- plan flows in a non-blocking fashion with get-next iterators
- Note some assumptions folded in here:
  - selections are "pushed down"
  - projections are "pushed down"
  - all this is only for single query blocks
- Some other popular assumptions (System R)
  - only left-deep plan trees
  - avoid Cartesian products

- Cost Estimation
  - The soft underbelly of query optimization.
  - Requires:
    - estimation of costs for each operator based on input cardinalities
    - both I/O & CPU costs to some degree of accuracy
    - estimation of predicate selectivities to compute cardinalities for next step
    - assumption of independence across predicates
    - decidedly an inexact science.
    - "Selingerian" estimation (no longer state of the art, but not really so far off.)
      - # of tuples & pages
      - # of values per column (only for indexed columns)
      - These estimations done periodically (why not all the time?)
      - back of envelope calculations: CPU cost is based on # of RSS calls, no distinction between Random and Sequential IO
      - when you can't estimate, use the "wet finger" technique
  - New alternative approaches:
    - Sampling: so far only concrete results for base relations
    - Histograms: getting better. Common in industry, some interesting new research.
    - Controlling "error propagation"
    - Robust plans (error-tolerant)

- Searching the Plan Space
  - Exhaustive search
  - Dynamic Programming (prunes useless subtrees): System R
  - Top-down, transformative version of DP: Volcano, Cascades (used in MS SQL Server?)
  - Randomized search algorithms (e.g. Ioannidis & Kang)
  - Job Scheduling techniques
  - In previous years we read many of these (they're fun!), but it is arguably more relevant to talk about query rewriting.
  - The System R Optimizer’s Search Algorithm
Look only at left-deep plans: there are $n!$ plans (not factoring in choice of join method)

Observation: many of those plans share common prefixes, so don’t enumerate all of them

Sounds like a job for … Dynamic Programming!

- Find all plans for accessing each base relation
- Include index scans when available on "SARGable" predicates
- For each relation, save cheapest unordered plan, and cheapest plan for each "interesting order". Discard all others.
- Now, try all ways of joining all pairs of 1-table plans saved so far. Save cheapest unordered 2-table plans, and cheapest "interesting ordered" 2-table plans.
- note: secondary join predicates are just like selections that can’t be pushed down
- Now try all ways of combining a 2-table plan with a 1-table plan. Save cheapest unordered and interestingly ordered 3-way plans. You can now throw away the 2-way plans.
- Continue combining $k$-way and 1-way plans until you have a collection of full plan trees
- At top, satisfy GROUP BY and ORDER BY either by using interestingly ordered plan, or by adding a sort node to unordered plan, whichever is cheapest.

Some additional details:
- don’t combine a $k$-way plan with a 1-way plan if there’s no predicate between them, unless all predicates have been used up (i.e. postpone Cartesian products)
- Cost of sort-merge join takes existing order of inputs into account.

Evaluation:
- Only brings complexity down to about $n2^{n-1}$, and you store $\binom{n}{n/2}$ plans
- But no-Cartesian-products rule can make a big difference for some queries.
- For worst queries, DP dies at 10-15 joins
- adding parameters to the search space makes things worse (e.g. expensive predicates, distribution, parallelism, etc.)

Simple variations to improve plan quality:
- bushy trees: $(2(n-1))/(n-1)!$ plans, DP complexity is $3^n - 2^{n+1} + n + 1$ need to store $2^n$ plans (actually it’s worse for subtle reasons)
- consider cross products: maximizes optimization time