Lecture 3 - IBM Papers

Access Path Selection (Selinger)

Intro, setup

- System R being developed since 1975; this paper is from 1979
- This is the bible of query optimization
- Remember, coming from the CODASYL era, so some of this is non-obvious:
  - A user does not need to provide join paths (remember nested loopy-approach?)
  - A user does not need to decide how each table is accessed
  - And the system figures out how to optimize to minimize cost
- What was known about query optimization at this point?
  - "Agile" First cut:
    - Divide and Conquer
      - Break complex expressions into subproblems and optimize those.
      - INGRES's *one-variable query processor* (DECOMP)
      - Strategy:
        - Pick the smallest table.
        - For each tuple, recursively evaluate the remainder of the query.
          - Essentially nested loops beginning with smallest relation as outer loop
    - Transformation Rules
      - Push filters before joins. Filters are cheap.
- Paper asks: *what if you could build an oracle?*
  - Moving query processing from programming to specification
  - There is a **finite** space of possible query plans.
  - Optimization becomes **search**.
    - Enumerate all plans.
    - Choose the one that minimizes an objective *in the given context*.
- Essentially divides QO into three main problems
  - Define the **search space** (with pruning heuristics).
  - Build a **noisy cost predictor**.
  - Design an **intelligent search algorithm**.
- These can be improved *independently* — and the community has been doing exactly that for ~50 years.
- The query optimization process in System R involves four steps, pretty reasonable:
  - parsing - also decomposition into query blocks (SFW)
    - mostly checking sytnax
  - optimization -
    - first looks up tables and columns and verifies existence -- all via the catalog
    - identifies possible access paths, statistics
    - then selects access paths (aka optimizes)
      - does so per query block, after determining which query block needs to execute first
      - per query block, permutations of joins and which join is selected
    - output in some IR they call
  - code generation
    - maps each physical operator into code
    - the code involves accesses to the RSS (relational storage system)
      - remember breakdown from lecture 1 on query processor vs. storage manager?

Storage system:

- maintains tables, provides access paths (indexes), maintains buffers, locking (CC) - only method of CC at the time, logging (durability)
- provides "tuples at a time" - essentially "volcano model"
- access paths:
  - sequential scan (called a segment scan in this paper), one page at a time, return all tuples of the relation
    - here, a segment comprising a set of pages can have more than one relation - but as storage has become cheap, it's easier to more clearly demarcate pages of one relation from another
  - index scan, impl. via B+trees, returns tuples from leaf nodes
    - clustered if data is sorted in index order.
    - Q: why can an index sometimes be worse than a sequential scan?
  - both accept SARGs - eliminating tuples before provided as a result to the next operator
    - implicitly doing predicate pushdown

Cost estimation:

- Overall cost = pages read (IO) + α . tuples processed (Compute)
- Dominant cost in 1979 was IO
- Statistics include
  - Table:
    - Cardinality, # pages, (also fraction of a segment that is for a given table)
      - For indexed attributes: # of distinct values, size of index, among others
    - Q: What other stats would be helpful?
- Using these, a "selectivity" is assigned to each predicate
  - Some somewhat reasonable, eg uniform distribution, independence of predicates
  - some pretty funny 1/3, 1/4
  - but easy to see how this can go pretty wrong already Q: thoughts?
    - Skew, correlations,
- Good news: we don't need exact costs — just **correct rankings**
- Card for an expression (aka the RSICARD in paper) = product of cardinalities of relations and the predicate selectivities
- Physical properties of interest to track: sorted order, called "interesting order"
  - orders for ORDER BY, GROUP BY, join attributes,
- Cost for various physical operators:
  - Clustered index lookup: $selectivity \times (\#indexpages + \#relpages) + \alpha \times RSICARD$
  - Unclustered index lookup: $selectivity \times (\#indexpages + \#relcardinality) + \alpha \times RSICARD$
  - Mostly textbook stuff!

Join Implementations and Order Selection:

- Nested loops
- Sort merge join
  - produces an interesting order AND benefits from interesting orders
- Note: hash joins had not been invented yet! 1980s
  - requires more memory
- n! orders
- But joining the n+1th relation to the output of the first n, is independent of "how" the first n is done: optimal substructure => bottom-up Dynamic Programming
  - also known as Bellman's *Principle of Optimality*

    - DP extended with context (interesting order)
- Essentially, for each subset of relations to be joined, and each interesting order
    - maintain the best result
    - 2^n plans x number of interesting orders
- Restrictions in paper:
    - Cartesian products as late as possible
        - Q: Why is this generally a good idea?
        - Q: When can this be bad?
    - Left-deep join trees - no bushy or right-deep
        - Bushy would increase complexity to 3^n
    - ORDER BY and GROUP BY last
        - Q: Why a bad idea? GROUP BY in particular can reduce result size considerably

Nested queries

- Scalar subqueries (or even uncorrelated subqueries involving IN etc) are evaluated eagerly
    - `SELECT NAME FROM EMPLOYEE WHERE SALARY = (SELECT AVG(SALARY) FROM EMPLOYEE)`
- Correlated subquery example:
    - `SELECT NAME FROM EMPLOYEE X WHERE SALARY > (SELECT SALARY FROM EMPLOYEE WHERE EMPLOYEE_NUMBER= X.MANAGER)`
- "Giant waving of hands"
    - either evaluate per outer tuple; and
    - cache query results; and/or
    - sort intermediates to make "inner" query evaluation cheap
    - future work: rewrite into joins!

Takeaways

- This is a great example of top-down systems research:
    - Start with the *whole problem*.
    - Factor it carefully.
    - Then iterate on components
- Databases as a Cross-Cut of CS
    - This is a classic “AI” problem:
        - Search (DP)
        - Statistical prediction
    - Decades before “AI” was fashionable.
- Database research is a license to do **computer science**, not just “systems” or “theory”.
- Some problems stay useful *and open* for decades.

Granularity of Locks and Degrees of Consistency (1976)

- This is actually two papers; one somewhat flawed

Lock Granularity

Setup