Recap

End-to-End Optimization

- **Input: SQL**
  - SQL is converted directly to something resembling relational algebra
  - Some DBs (e.g., Postgres) use a more complex structure that represents a joint cross-product, selection, and projection
- **Naive RA**
  - Some RA rewrites can be applied to RA to produce guaranteed faster plans
    - Selection Pushdown
    - Join Conversion
    - In some situations, Projection pushdown may also help
    - Eliminating redundant "Distinct" operators
    - Eliminating redundant "Sort" operators
  - These operations are applied to a "fixed point"
    - As long as an opportunity exists to apply the optimization, it is applied
  - The output of this stage is just another RA tree
- **Optimized RA**
  - The system next explores rewrites that do not guarantee better performance
    - Different Join Orders
    - Different Access Paths
  - The system builds an execution plan for each possibility
    - A plan also "decorates" the RA plan, noting the specific algorithm used to implement it.
  - The system estimates the cost of each possible plan

Overview

- **How do we estimate IO Cost?**
  - Number of reads performed by each operator
  - Number of writes performed by each operator
- **What about communicating between operators?**
  - Assume operators can communicate with each other for free.
  - Costs only include:
    - The cost of materializing the data IF it needs to be materialized on disk
    - The cost of reading the data back in IF it needs to be read back in.
- **What else do we need?**
  - For some of these estimates, we'll need to be able to estimate the size of each table (call the # of pages in R: $|R|$)
  - Basic properties of the data:
    - Key Columns
    - Distribution of Values

IO Costs

- **File Scan (R)**
  - Number of IOs : $|R|$
- **Index Lookup (σ(R) where R is a file scan)**
  - Number of IOs for a Hash Index : $|\sigma(R)|$
    - How big is this? Return to it later.
  - Number of IOs for a B+Tree Index with directory pages of size B: $|\sigma(R)| + \log_B(|R|)$
- **Selection (σ(R))**
  - Number of IOs : 0 (never need to materialize a selection)

- **Projection (π(R))**
  - Number of IOs : 0 (never need to materialize a projection)
- **Union**
  - Number of IOs : 0 (never need to materialize a BAG union — see distinct for set union)
- **Sort (τ(R)) — External Sort with B pages of memory**
  - Number of IOs : $\sim 2 \cdot \log_B(|R| / 2)$
- **Cross-Product (R x S) — BNLJ with B pages of memory for blocking R**
  - Number of IOs : |S| + (|R| / B)•(|S|)
    - Need to write all of S to disk once: |S| pages
    - Repeat (|R| / B) times…
      - Read B pages of data from source operator R: Free
      - Join the block with the materialized data in S, one tuple at a time: |S|

More IO Costs

Join (R ⋈ S) — 1-pass Hash/Tree Join

- **Number of IOs: 0 (entirely in-memory)**

Join (R ⋈ S) — 2-pass Hash Join

- **Number of IOs: 2•(|R| + |S|)**
  - Write all |R| and |S| to disk, bucketizing: |R| + |S|
  - Read in each bucket: |R| + |S|

Join (τ(R) ⋈ τ(S)) — Sort/Merge Join

- **Number of IOs: 0 + cost of the τ(S) (Merge step is free)**

Join (R $\bowtie_{R.A = S.A}$ S) — Index Nested Loop Join (assuming index on S)

- **Number of IOs: |R| • [cost of one index lookup: $\sigma_{[const] = S.A}$(S)]**
  - Each inner loop is basically one Index Scan

Aggregation (ɣ(R)) — In-memory

- **Number of IOs: 0**

Aggregation (ɣ(R)) — On-Disk, Hash-Based

- **Number of IOs: 2|R|**
  - Write each bucket out, read each bucket in

Aggregation (ɣ(τ(R)) — On-Disk, Sort-Based

- **Number of IOs: 0 + cost of τ(R)**

Distinct (δ(R))— Works EXACTLY like Aggregation

Cardinality (Size) Estimation

Most of the operators are straightforward

- **π(R), τ(R) : |R|**
- **R U S : |R| + |S|**
- **R x S : |R| * |S|**
- **R ⋈ S : Identical to σ(R x S)…**

Some are hard

- **σ(R)**
- **ɣ(R) & δ(R)**

- Selection : Compute Selectivity (or % tuples passed through)
  - **Generic (Default) Heuristic:**
    - Selectivity = 0.5
    - Works … mostly well 70% of the time. Very brittle and liable to break things
    - **Be wary**: DBMSes actually do this!
  - **R.A = [Const]**
    - If R.A is a Key, then precisely 1 tuple passes through… given
    - **Idea**: Collect stats: # of distinct values
      - Selectivity = 1 / # of distinct values of R.A
      - Works well… but only for discrete data (Strings, Ints, Dates)
      - Also gives you "Key" for free
      - Also works for R.A in [List]
  - **R.A < [Const] (also works for others)**
    - **Idea**: Collect stats: Min/Max, and assume a uniform distribution of values
      - Selectivity = ([Const] - Min) / (Max - Min)
      - Works for continuous data (Floats)
  - **R.A = R.B**
    - (the Equijoin condition)
    - **Idea 1**: Assume no correlation
      - Becomes identical to either R.A = const or R.B = const
      - For each row, you're testing whether R.B = Some specific, somewhat arbitrary value
      - **Both** are an upper bound on the selectivity, so take whichever reduction gives you the lower value
  - **C1 AND C2**
    - Assuming no correlation between C1 and C2: Selectivity(C1) • Selectivity(C2)
- Going more fancy: Histograms (See attached)