Recap: Query Processing

- Many, many ways to implement the relational operations
  - Numerous more used in practice
  - Especially in data warehouses which handles TBs (even PBs) of data
- However, SQL is complex, and you can do much with it
  - Compared to that, this isn't much
- Most of it is very nicely modular
  - Especially through use of the iterator() interface
  - Can plug in new operators quite easily
  - PostgreSQL query processing codebase very easy to read and modify
- Having many operators does complicate the query optimizer
  - But needed for performance

Query Optimization

- Overview
- Statistics Estimation
- Transformation of Relational Expressions
- Optimization Algorithms

Query Optimization

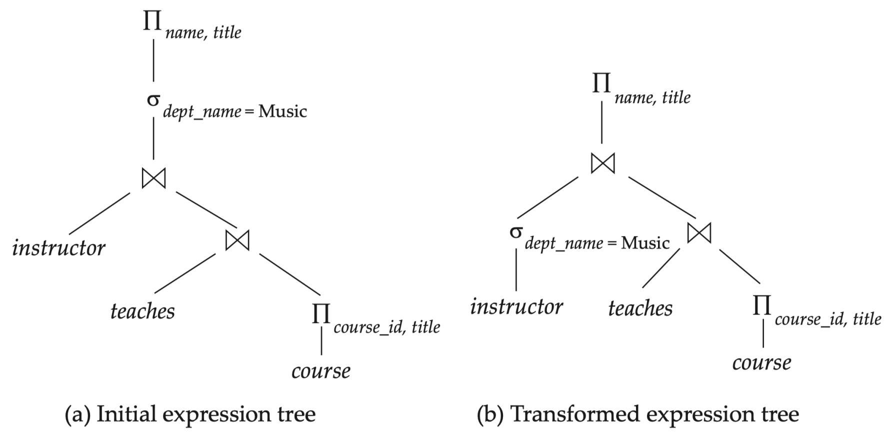
- Why ?
  - Many different ways of executing a given query
  - Huge differences in cost
- Example:
  - select * from person where ssn = "123"
  - Size of person = 1GB
  - Sequential Scan:
    - Takes 1GB / (20MB/s) = 50s
  - Use an index on SSN (assuming one exists):
    - Approx 4 Random I/Os = 16ms

Query Optimization

- Many choices
  - Using indexes or not, which join method (NL, hash, merge…)
  - What join order ?
    - Given a join query on R, S, T, should I join R with S first, or S with T first ?
- This is an optimization problem
  - Similar to say traveling salesman problem
  - Number of different choices is very very large
  - Step 1: Figuring out the solution space
  - Step 2: Finding algorithms/heuristics to search through the solution space

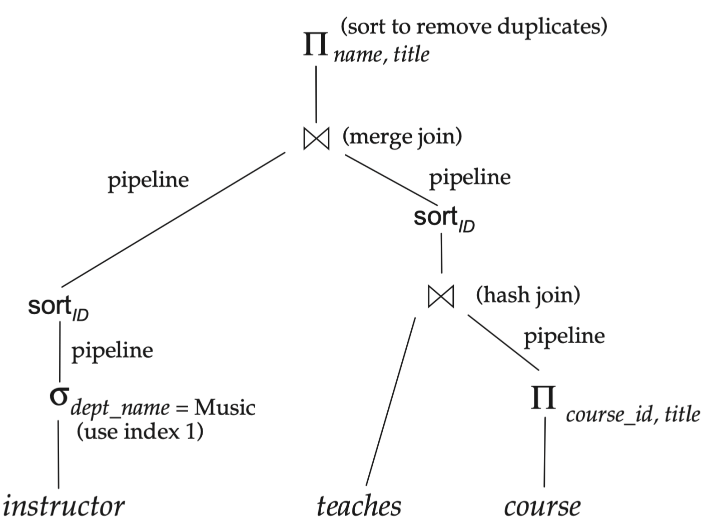
Query Optimization

- Equivalent relational expressions
  - Drawn as a tree
  - List the operations and the order
    - note that the select operator has moved



Query Optimization

- Query evaluator internally annotates expressions annotated with the methods to be used



Query Optimization

- Steps:
  - Generate all possible execution plans for the query
  - Figure out the cost for each of them
  - Choose the best

- Not done exactly as listed above
  - Too many different execution plans for that
  - Typically interleave all of these into a efficient search algorithm

Query Optimization

- Steps (detail):
  - Generate all possible execution plans for the query
    - First generate all equivalent expressions
    - Then consider all annotations for the operations
  - Figure out the cost for each of them
    - Compute cost for each operation
      - Using the formulas discussed before
      - One problem: How do we know the number of result tuples?
    - Count them ! Better yet, estimate…
  - Choose the lowest estimate…

Query Optimization

- Introduction
- Transformation of Relational Expressions
- Statistics Estimation
- Optimization Algorithms

Equivalence of Expressions

- Two relational expressions equivalent iff:
  - Their result is identical on all *legal* databases
- Equivalence rules (Section 16.2):
  - Allow replacing one expression with another
- Examples:

1. $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$

2. Selections are commutative

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

Equivalence Rules

- Examples:

  3. $\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) \equiv \Pi_{L_1}(E)$ *if L_1 is subset of L_2 etc.*

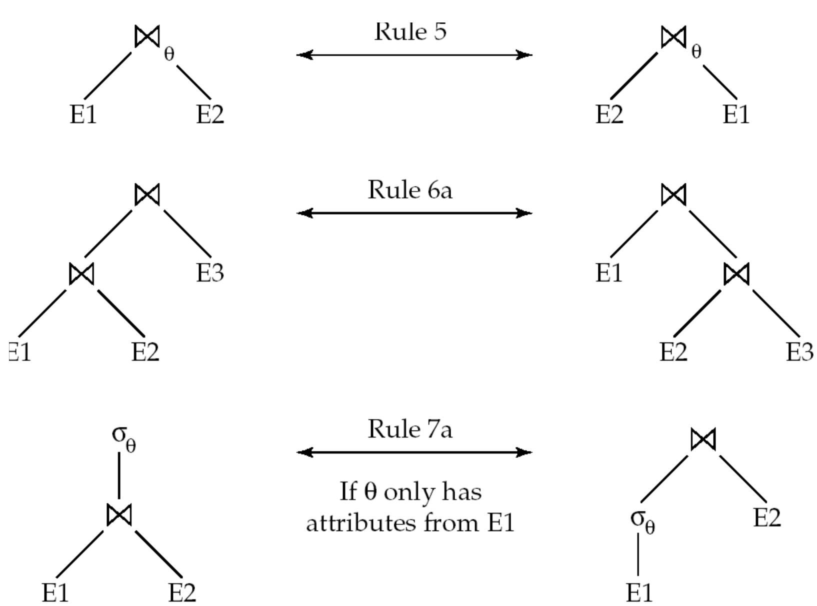
  5. $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$

  7(a). If θ_0 only involves attributes from E_1:

  $$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

- And so on…
  - Many rules of this type

Pictorial Depiction



Natural, inner joins commutative

Natural, inner joins associative

several more

Example

- Find the names of all customers with an account at a Brooklyn branch whose account balance is over $1000.

$$\Pi_{customer\_name}(\sigma_{branch\_city = \text{"Brooklyn"} \wedge balance > 1000}(branch \bowtie (account \bowtie depositor)))$$

- Apply the rules one by one

$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{"Brooklyn"} \wedge balance > 1000}(branch \bowtie account)) \bowtie depositor)$$

$$\Pi_{customer\_name}(((\sigma_{branch\_city = \text{"Brooklyn"}}(branch)) \bowtie (\sigma_{balance > 1000}(account))) \bowtie depositor)$$

first predicate on ***branch***

second predicate on ***account***

Equivalence of Expressions

- The rules give us a way to enumerate all equivalent expressions
  - Note that the expressions don't yet contain physical access methods, join methods etc…
- Simple Algorithm:
  - Start with the original expression
  - Apply all possible applicable rules to get a new set of expressions
  - Repeat with this new set of expressions
  - Till no new expressions are generated

Evaluation Plans

- We still need to choose the join methods etc..
  - Option 1: Choose for each operation separately
    - Usually okay, but sometimes the operators interact
    - Consider joining three relations on the same attribute:
      - $R1 \bowtie_a (R2 \bowtie_a R3)$
    - Best option for R2 join R3 might be hash-join
      - But if *R1* is sorted on *a,* then *sort-merge join* preferable because it produces the result in sorted order by *a*
- Also, pipelining or materialization
- Such issues typically arise when doing the optimization

Query Optimization

- Introduction
- Example of a Simple Type of Query
- Transformation of Relational Expressions
- Optimization Algorithms
- Statistics Estimation

Optimization Algorithms

- Two types:
  - Exhaustive: attempt to find the best plan
  - Heuristic: simpler, but not guaranteed to find the optimal plan
- Consider a simple case
  - Join of the relations *R1, …, Rn*
  - No selections, no projections
- Still very large plan space

Searching for the best plan

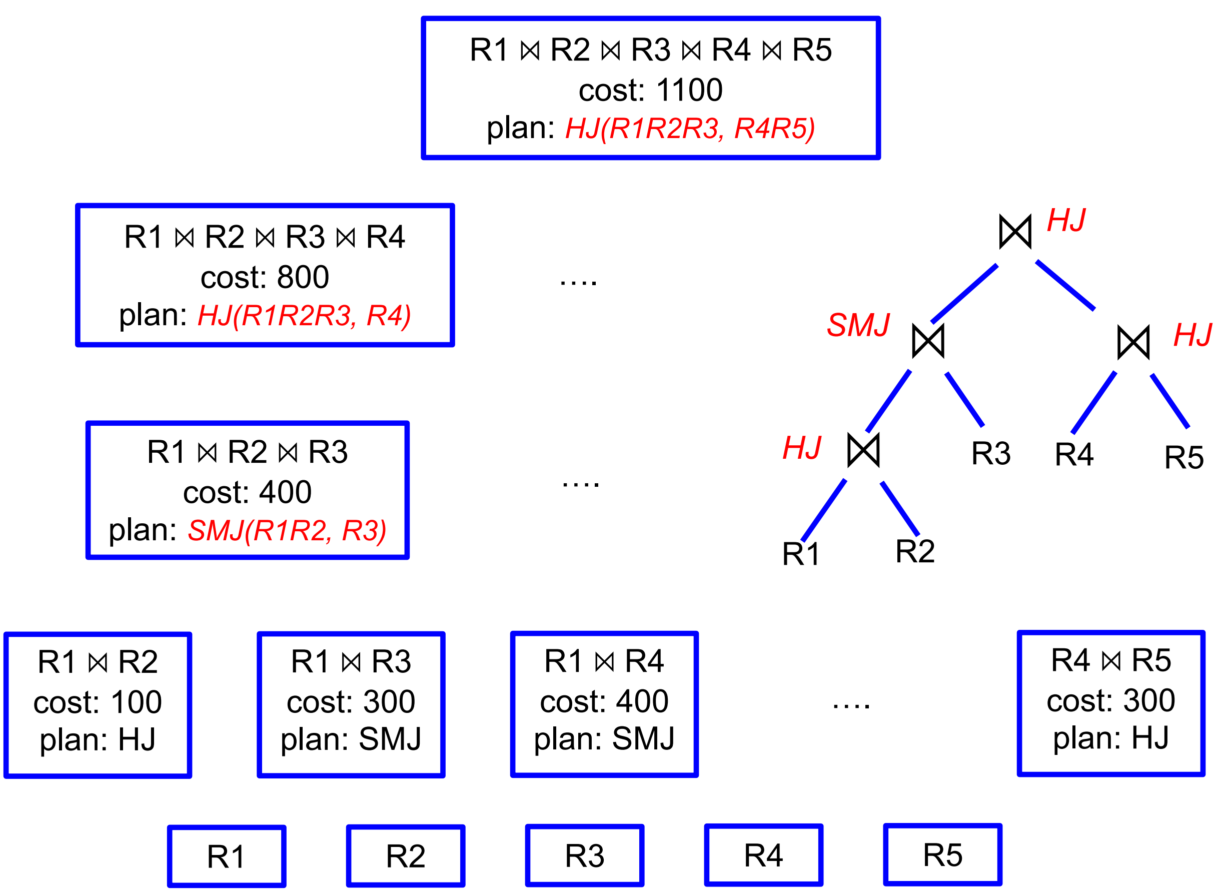
Option 1:

- Works! …but is not feasible
- Consider a simple case:
  - *R1 ⋈ (R2 ⋈ (R3 ⋈ (… ⋈ Rn)))….)*
- Just join commutativity and associativity will give us:
  - At least:
    - $n^2 * 2^n$
  - At worst:
    - $n! * 2^n$ (factorial results from linear join order)
  - *Enumeration usually combined into a directed search*

Searching for the best plan

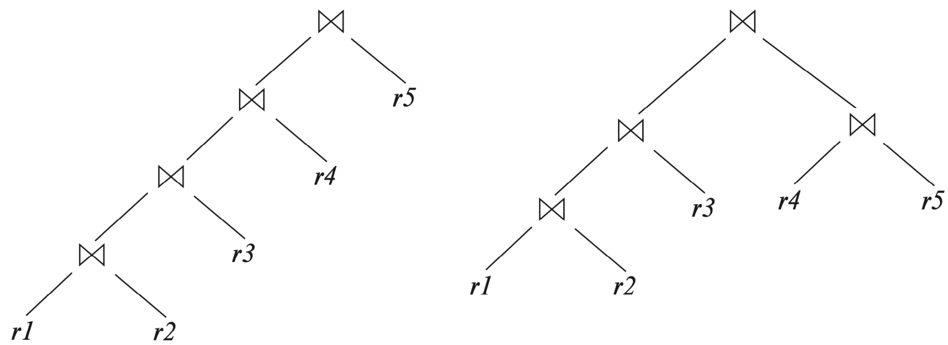
Option 2:

- Dynamic programming
  - There is much commonality between the plans
  - Costs are additive
    - Caveat: Sort orders (also called “interesting orders”)
- Reduces costs to O($n3^n$) or O($n2^n$) in most cases
  - Interesting orders increase this a little bit
- Considered acceptable
  - Typically $n < 10$.
- Switch to heuristic if not acceptable



Left Deep Join Trees

- In left-deep join trees, the right-hand-side input for each join is a relation, not the result of an intermediate join
- Early systems only considered these types of plans
  - Easier to pipeline



(a) Left-deep join tree (b) Non-left-deep join tree

Heuristic Optimization

- Dynamic programming is expensive
- Use *heuristics* to reduce the number of choices
- Typically rule-based:
  - Perform selection early (reduces num tuples for later ops)
  - Perform projection early (reduces num `attributes)
  - Perform most restrictive selection and join operations before other similar operations.
- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Steps in Typical Heuristic Optimization

Equiv rules in 16.2.1

1. Deconstruct conjunctive selections into a sequence of single selection operations (Equiv. rule 1.).
2. Move selection operations down the query tree for the earliest possible execution (Equiv. rules 2, 7a, 7b, 11).
3. Execute first those selection and join operations that will produce the smallest relations (Equiv. rule 6).
4. Replace Cartesian product operations that are followed by a selection condition by join operations (Equiv. rule 4a).
5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed (Equiv. rules 3, 8a, 8b, 12).
6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining).

Query Optimization

- Introduction
- Example of a Simple Type of Query
- Transformation of Relational Expressions
- Optimization Algorithms
- Statistics Estimation

Cost estimation

- Computing operator costs requires information like:
  - Primary key ?
  - Sorted or not, which attribute
    - So we can decide whether need to sort again
  - How many tuples in the relation, how many blocks ?
  - RAID ?? Which one ?
    - Read/write costs are quite different
  - How many tuples match a predicate like "age > 40" ?
    - E.g. Need to know how many index pages need to be read
  - Intermediate result sizes
    - E.g. (R JOIN S) is input to another join operation – need to know if it fits in memory
  - And so on…

Cost estimation

- Some info is static and maintained in the metadata
  - Primary key ?
  - Sorted or not, which attribute
    - So we can decide whether need to sort again
  - How many tuples in the relation, how many blocks ?
  - RAID ?? Which one ?
    - Read/write costs are quite different

- Typically kept in some tables in the database
  - "all_tab_columns" in Oracle
  - Postgresql: `analyze` cmd updates `pg_statistic` and `pg_stats`
- Most systems have commands for updating them

Cost estimation

- Others need to be estimated:
  - How many tuples match a predicate like "age > 40" ?
    - E.g. Need to know how many index pages need to be read
  - Intermediate result sizes
- The problem variously called:
  - "intermediate result size estimation"
  - "selectivity estimation"

- Very important to estimate reasonably well
  - e.g. consider "SELECT * FROM R WHERE zipcode = 20742"
  - We estimate that there are 10 matches, and choose to use a secondary index (remember: random I/Os)
  - If turns out there are 10000 matches
    - using a secondary index very bad idea…
  - Optimizer often choose block-nested-loop joins if one relation very small

… underestimation can be very bad

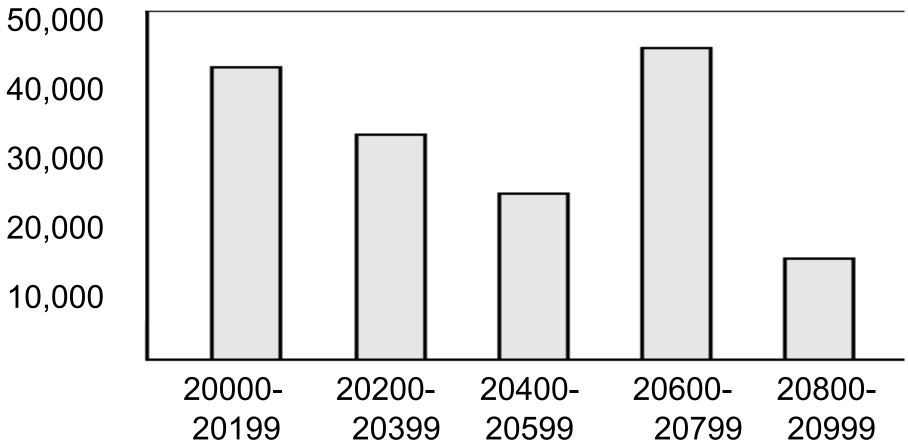
Selectivity Estimation

- Basic idea:
  - Maintain some information about the tables
    - More information → more accurate estimation
    - More information → higher storage cost, higher update cost
  - Make uniformity and randomness assumptions to fill in the gaps

- Example:
  - For a relation "people", we keep:
    - Total number of tuples = 100,000
    - Distinct "zipcode" values that appear in it = 100
  - Given a query: "zipcode = 20742"
    - We estimated the number of matching tuples as: 100,000/100 = 1000
  - What if I wanted more accurate information ?
    - Keep histograms…

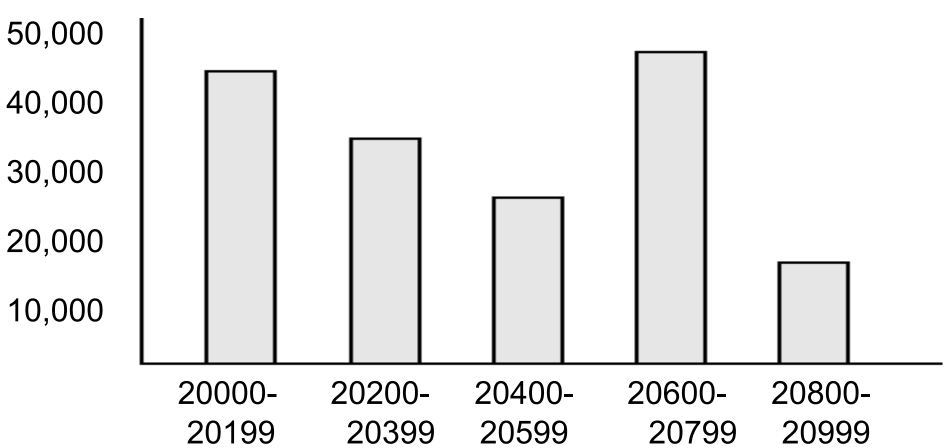
Histograms

- A condensed, approximate version of the "frequency distribution"
  - Divide the range of the attribute value in "buckets"
  - For each bucket, keep the total count
  - Assume uniformity within a bucket



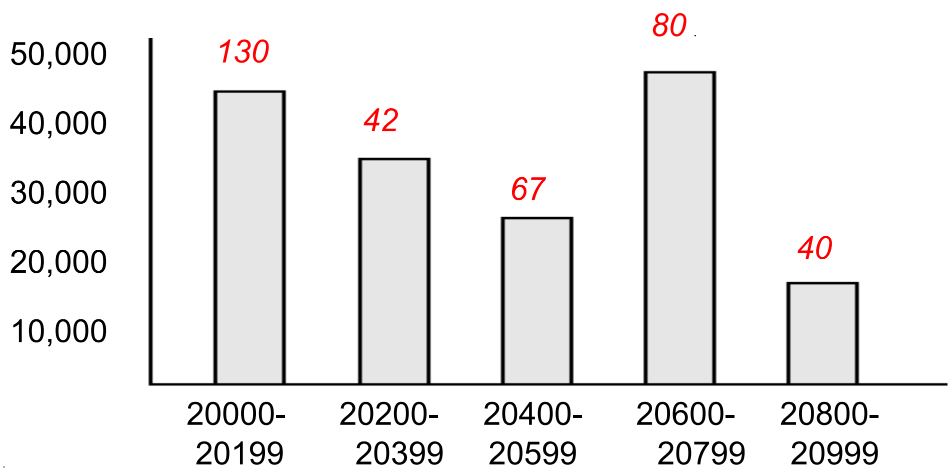
Histograms

- Given a query: zipcode = " 20742"
  - Find the bucket (Number 3)
  - Say the associated count = 45000
  - Assume uniform distribution within the bucket: 45,000/200 = 225



Histograms

- What if the ranges are typically not full ?
  - ie., only a few of the zipcodes are actually in use ?
- With each bucket, also keep the number of distinct values used for zipcodes
- Now the estimate would be: 45,000/80 = 562.50
- More Information → Better estimation



Exam #2

- Functional dependences (extraneous attributes, covers)
- Storage manager
- RAID
- File organization (heap, sorted, hash)
- Indexes (primary / secondary, dense sparse, hash)
  - B+-trees: height, cost of access, including xtra leaves
  - insertions, deletions
- Query execution (including costs)
  - selections
  - joins (block nested, hash, merge, index nested..)
  - sorts (in-memory, external)
- Query estimation
  - histograms
  - uniformity
  - using attribute stats
- Query optimization
  - execution trees
  - materialization/pipelining