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 \texttt{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

Databases

- Data Models
  - Conceptual representation of the data
- Data Retrieval
  - How to ask questions of the database
  - How to answer those questions
- Data Storage
  - How/where to store data, how to access it
- Data Integrity
  - Manage crashes, concurrency
  - Manage semantic inconsistencies
Query Optimization

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

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 \( \frac{1GB}{(20MB/s)} = 50s \)
- Use an index on SSN (assuming one exists):
  - Approx 4 Random I/Os = 40ms
Query Optimization

- Many choices
  - Using indexes or not, which join method (hash, vs merge, vs NL)
  - 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
Query Optimization

- Execution plans
  - Evaluation expressions annotated with the methods used

```
Π_{customer_name} (sort to remove duplicates)

(hash join)

(merge join) depositor

pipeline

σ_{branch_city = Brooklyn}
(use index 1) branch

pipeline

σ_{balance < 1000}
(use linear scan) account
```
Query Optimization

- **Steps:**
  - 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 for, say, \( \text{balance} < 2500 \) (account)
    - Count them! Better yet, estimate…
  - Choose the best

---

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

- 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)
  - Turns out there are 10000 matches
  - Using a secondary index very bad idea
  - Optimizer also often choose 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

```
<table>
<thead>
<tr>
<th>Bucket</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>20000-20199</td>
<td>50,000</td>
</tr>
<tr>
<td>20200-20399</td>
<td>40,000</td>
</tr>
<tr>
<td>20400-20599</td>
<td>30,000</td>
</tr>
<tr>
<td>20600-20799</td>
<td>20,000</td>
</tr>
<tr>
<td>20800-20999</td>
<td>10,000</td>
</tr>
</tbody>
</table>
```

**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?
  - i.e., only a few of the zipcodes are actually in use?
- With each bucket, also keep the number of zipcodes that are valid
- Now the estimate would be: 45,000/80 = 562.50
- More Information → Better estimation

Histograms

- Very widely used in practice
  - One-dimensional histograms kept on almost all columns of interest
    - i.e., the columns that are commonly referenced in queries
  - Sometimes: multi-dimensional histograms also make sense
    - Less commonly used as of now
- Two common types of histograms:
  - Equi-depth
    - The attribute value range partitioned such that each bucket contains about the same number of values
  - Equi-width
    - The attribute value range partitioned in equal-sized buckets
  - others…
Next...

- Estimating sizes of the results of various operations
- Guiding principle:
  - Use all the information available
  - Make uniformity and randomness assumptions otherwise
  - Many formulas, but not very complicated...
    - In most cases, the first thing you think of

Basic statistics

- Basic information stored for all relations
  - $n_r$: number of tuples in a relation $r$.
  - $b_r$: number of blocks containing tuples of $r$.
  - $l_r$: size of a tuple of $r$.
  - $f_r$: blocking factor of $r$ — i.e., the number of tuples of $r$ that fit into one block.
  - $V(A, r)$: number of distinct values that appear in $r$ for attribute $A$; same as the size of $\Pi_A(r)$.
  - $\text{MAX}(A, r)$: th maximum value of $A$ that appears in $r$
  - $\text{MIN}(A, r)$
  - If tuples of $r$ are stored together physically in a file, then:

$$b_r = \frac{n_r}{f_r}$$
Selection Size Estimation

- $\sigma_{A=v}(r)$
  - $n_r / V(A,r)$: number of records that will satisfy the selection
  - equality condition on a key attribute: size estimate = 1

- $\sigma_{A\geq r}(r)$ (case of $\sigma_{A\geq V}(r)$ is symmetric)
  - Let $c$ denote the estimated number of tuples satisfying the condition.
  - If $\min(A,r)$ and $\max(A,r)$ are available in catalog
    - $c = 0$ if $v < \min(A,r)$
    - $c = n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - If histograms available, can refine above estimate
  - In absence of statistical information $c$ is assumed to be $n_r / 2$.

Size Estimation of Complex Selections

- selectivity$(\theta_j) = \text{the probability that a tuple in } r \text{ satisfies } \theta_j$.
  - If $s_j$ is the number of satisfying tuples in $r$, then selectivity $(\theta_j) = s_j/n_r$

- conjunction: $\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n}(r)$. Assuming independence, estimate of tuples in the result is:
  $$n_r \cdot \frac{S_1 \cdot S_2 \cdot \ldots \cdot S_n}{n_r^n}$$

- disjunction: $\sigma_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n}(r)$. Estimated number of tuples:
  $$n_r \cdot \left(1 - \left(1 - \frac{S_1}{n_r}\right) \cdot \left(1 - \frac{S_2}{n_r}\right) \ldots \cdot \left(1 - \frac{S_n}{n_r}\right)\right)$$

- negation: $\sigma_{\neg \theta}(r)$. Estimated number of tuples: $n_r - \text{size}(\sigma_\theta(r))$
Joins

- **R JOIN S: R.a = S.a**
  - |R| = 10,000; |S| = 5000

- **CASE 1: a is key for S**
  - Each tuple of R joins with exactly one tuple of S
  - So: |R JOIN S| = |R| = 10,000

  - What if there is a selection on R or S
    - Adjust accordingly
    - Say: S.b = 100, with selectivity 0.1
    - THEN: |R JOIN S| = |R| * 0.1 = 100

- **CASE 2: a is key for R**
  - Similar

- **CASE 3: a is not a key for either**
  - Reason with the distributions on a
  - Say: the domain of a: V(A, R) = 100 (the number of distinct values a can take)
  - THEN, assuming uniformity
    - For each value of a
      - We have 10,000/100 = 100 tuples of R with that value of a
      - We have 5000/100 = 50 tuples of S with that value of a
      - All of these will join with each other, and produce 100 *50 = 5000
    - So total number of results in the join:
      - 5000 * 100 = 500,000
  - We can improve the accuracy if we know the distributions on a better
    - Say using a histogram
Other Operations

- **Projection:** $\Pi_A(R)$
  - If no duplicate elimination, THEN $|\Pi_A(R)| = |R|$.
  - If *distinct* used (duplicate elimination performed): $|\Pi_A(R)| = V(A, R)$.

- **Set operations:** (heuristic upper bounds)
  - Union ALL: $|R \cup S| = |R| + |S|$.
  - Intersect ALL: $|R \cap S| = \min\{|R|, |S|\}$.
  - Except ALL: $|R - S| = |R|$.
  - Union, Intersection, Except (with duplicate elimination)
    - Somewhat more complex reasoning based on the frequency distributions etc…

- And so on …

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:
  - Allow replacing one expression with another
- Examples:
  1. \( \sigma_{\theta_1 \land \theta_2} (E) = \sigma_{\theta_1} (\sigma_{\theta_2} (E)) \)
  2. Selections are commutative
     \( \sigma_{\theta_1} (\sigma_{\theta_2} (E)) = \sigma_{\theta_2} (\sigma_{\theta_1} (E)) \)

Equivalence Rules

- Examples:
  3. \( \Pi_{L_1} (\Pi_{L_2} (\ldots (\Pi_{L_n} (E)) \ldots)) = \Pi_{L_1} (E) \)
  5. \( E_1 \bowtie_{\theta_0} E_2 = E_2 \bowtie_{\theta_0} E_1 \)
  7(a). If \( \theta_0 \) only involves attributes from \( E_1 \)
     \( \sigma_{\theta_0} (E_1 \bowtie_{\theta_0} E_2) = (\sigma_{\theta_0} (E_1)) \bowtie_{\theta_0} E_2 \)

- And so on…
  - Many rules of this type
Example

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

  \[ \Pi_{\text{customer\_name}}(\sigma_{\text{branch\_city} = \text{“Brooklyn”} \land \text{balance > 1000}}(\text{branch} \bowtie (\text{account} \Join \text{depositor}))) \]

- Apply the rules one by one

  \[ \Pi_{\text{customer\_name}}((\sigma_{\text{branch\_city} = \text{“Brooklyn”} \land \text{balance > 1000}}(\text{branch} \bowtie \text{account})) \Join \text{depositor}) \]

  \[ \Pi_{\text{customer\_name}}(((\sigma_{\text{branch\_city} = \text{“Brooklyn”}}(\text{branch})) \Join (\sigma_{\text{balance > 1000}}(\text{account}))) \Join \text{depositor}) \]
Example

Equivalence of Expressions

- The rules give us a way to enumerate all equivalent expressions
  - Note that the expressions don’t 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
Equivalence of Expressions

- Works, but is not feasible
- Consider a simple case:
  - $R1 \Join (R2 \Join (R3 \Join \ldots \Join Rn))\ldots$

  - Just join commutativity and associativity will give us:
    - At least:
      - $n^2 \cdot 2^n$
    - At worst:
      - $n! \cdot 2^n$
    - Typically enumeration combined with the search process

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:
      - $R1 \Join_a (R2 \Join_a R3)$
    - Best option for R2 join R3 might be hash-join
      - But if $R1$ is sorted on $a$, then sort-merge join is 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 $R_1, \ldots, R_n$
  - No selections, no projections
  - Still very large plan space
Searching for the best plan

- **Option 1:**
  - Enumerate all equivalent expressions for the original query
    - Using the rules outlined earlier
  - Estimate cost for each and choose the lowest

- **Too expensive!**
  - Consider finding the best join-order for $r_1 \times r_2 \times \ldots \times r_n$.
  - There are $(2(n - 1))!/(n - 1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!

Searching for the best plan

- **Option 2:**
  - Dynamic programming
    - There is too much commonality between the plans
    - Also, 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
Dynamic Programming Algo.

- Join R1, R2, R3, R4, R5

Options:
1. Join R1R2 with R3 using HJ
   cost = 100 + cost of this join
2. Join R1R2 with R3 using SMJ
   cost = 100 + cost of this join
3. Join R1R3 with R2 using HJ
   cost = 300 + cost of this join

...
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

---

Heuristic Optimization

- Dynamic programming is expensive
- Use *heuristics* to reduce the number of choices
- Typically rule-based:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of 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

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
- Summary
Query Optimization

- Integral component of query processing
  - Why?
- One of the most complex pieces of code in a database system
- Active area of research
  - E.g. XML Query Optimization?
  - What if you don’t know anything about the statistics
  - Better statistics
  - Etc …