Query Processing

continued...

Hash-Join Iterator Interface

- **open():**
  - Call open() on the left and the right children
  - Decide if partitioning needed (if size of smaller relation > memory)
  - Create a hash table

- **get_next():** (no partitioning)
  - First call:
    - Get all tuples from the right child one by one (using get_next()), and insert them into the hash table
    - Read the first tuple from the left child (using get_next())
    - *Hash table includes the actual tuples*
  - All calls:
    - Probe into the hash table using the "current" tuple from the left child
      - Read a new tuple from left child if needed
    - Return exactly “one result”
      - Must keep track if more results need to be returned for that tuple
Hash-Join Iterator Interface

- **close()**: 
  - Call close() on the left and the right children 
  - Delete the hash table, other intermediate state etc…

- **get_next()**: (partitioning) 
  - First call: 
    - Get all tuples from both children and create the partitions on disk 
    - Read the first partition for the right child and populate the hash table 
    - Read the first tuple from the left child from appropriate partition 
  - All calls: 
    - Once a partition is finished, clear the hash table, read in a new partition from the right child, and re-populate the hash table 
  - Not that much more complicated

- Take a look at thepostgresql codebase

Pipelining (Cont.)

- In producer-driven or eager pipelining 
  - Operators produce tuples eagerly, pass to parents 
    - Buffer maintained between operators 
      - child puts tuples in buffer 
      - parent removes tuples from buffer 
    - if buffer is full: 
      - child waits till there is space in the buffer 
      - then generates more tuples 
  - System runs operations that have space in output buffer and can process more input tuples
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
- Most of it is very nicely modular
  - Especially through use of the iterator() interface
  - Can plug in new operators quite easily
  - PostgreSQL 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 $1\text{GB} / (20\text{MB/s}) = 50\text{s}$
  - 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 traveling salesman problem
  - Number of different choices is very large
  - Step 1: Map out the solution space
  - Step 2: Find algorithms/heuristics to search solution space
Query Optimization

- Equivalent relational expressions
  - Drawn as a tree
  - List the operations and the order

```
\begin{align*}
\Pi_{\text{customer\_name}} \\
\sigma_{\text{branch\_city}=\text{Brooklyn}} \\
\text{branch} \\
\text{account} \quad \text{depositor}
\end{align*}
```

Query Optimization

- Execution plans
  - Evaluation expressions annotated with the methods used

```
\begin{align*}
\Pi_{\text{customer\_name}} \text{ (sort to remove duplicates)} \\
\triangledown \text{ (hash join)} \\
\triangledown \text{ (merge join)} \\
\text{depositor} \\
\text{pipeline} \\
\sigma_{\text{branch\_city}=\text{Brooklyn}} \text{ (use index 1)} \\
\text{branch} \\
\text{pipeline} \\
\sigma_{\text{balance}<\text{1000}} \text{ (use linear scan)} \\
\text{account}
\end{align*}
```
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 single 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 for, say, \( \sigma_{\text{balance} \leq 2500}(\text{account}) \)?
    - 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 chooses 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

![Histogram Graph]

- 20000-
  - 20199
- 20200-
  - 20399
- 20400-
  - 20599
- 20600-
  - 20799
- 20800-
  - 20999

- 50,000
- 40,000
- 30,000
- 20,000
- 10,000
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?
  - 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 \( \Pi_A(r) \) (relational algebra semantics).
  - \( MAX(A, r) \): the maximum value of \( A \) that appears in \( r \).
  - \( MIN(A, r) \)
  - If tuples of \( r \) are stored together physically in a file, then:
    \[
    b_r = \left\lceil \frac{n_r}{f_r} \right\rceil
    \]

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

- \( \sigma_{A\leq v}(r) \) (case of \( \sigma_{A\geq v}(r) \) is symmetric)
  - Let \( c \) denote the estimated number of tuples satisfying the condition.
  - If \( \text{min}(A, r) \) and \( \text{max}(A, r) \) are available in catalog
    - \( c = 0 \) if \( v < \text{min}(A, r) \)
    - \( c = n_r \) if \( v \geq \text{max}(A, r) \)
    - \( c = n_r \frac{v - \text{min}(A, r)}{\text{max}(A, r) - \text{min}(A, r)} \)
  - 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**($\theta_i$) = the probability that a tuple in $r$ satisfies $\theta_i$.
  - If $s_i$ is the number of satisfying tuples in $r$, then selectivity ($\theta_i$) = $s_i/n_r$.

- **conjunction:** $\sigma_{\theta_1 \land \theta_2 \land \ldots \land \theta_n}(r)$. Assuming independence, estimate of tuples in the result is:
  \[ n_r \times \frac{s_1 \times s_2 \times \ldots \times s_n}{n_r^n} \]

- **disjunction:** $\sigma_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n}(r)$. Estimated number of tuples:
  \[ n_r \times \left(1 - \left(1 - \frac{s_1}{n_r}\right) \times \left(1 - \frac{s_2}{n_r}\right) \times \ldots \times \left(1 - \frac{s_n}{n_r}\right)\right) \]

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

we assume all predicates are independent

Size Estimation Examples

- Assume:
  - $n_r = 1000$
  - $\theta_1$ ("balance > $1000"$) = 0.9
  - $\theta_2$ ("age < 40") = 0.4
  - ("balance > $1000"$) AND ("age < 40")
    \[ = n_r \times \theta_1 \times \theta_2 \]
    \[ = 1000 \times 0.9 \times 0.4 = 360 \]
  - ("balance > $1000"$) OR ("age < 40")
    \[ = n_r \times (1 - (1 - \theta_1) \times (1 - \theta_2)) \]
    \[ = 1000 \times (1 - 0.1 \times 0.6) \]
    \[ = 1000 \times 0.94 \]
    \[ = 940 \]

we assume all predicates are independent