Query Processing

- Overview
- Selection operation
- Join operators
- Other operators
- Putting it all together…

Hash Join

- Case 1: Smaller relation (S) fits in memory
- Nested-loops join:
  
  \[
  \text{for each tuple } r \text{ in } R \\
  \quad \text{for each tuple } s \text{ in } S \\
  \quad \text{check if } r.a = s.a
  \]
- Cost: \(b_r + b_s\) transfers, 2 seeks
- The inner loop is not exactly cheap (high CPU cost)

- Hash join:
  
  \[
  \text{read } S \text{ in memory and build a hash index on it} \\
  \text{for each tuple } r \text{ in } R \\
  \quad \text{use the hash index on } S \text{ to find tuples such that } S.a = r.a
  \]
Hash Join

- **Case 1:** Smaller relation \((S)\) fits in memory

  Hash join:
  
  - **read** \(S\) **in memory** and build a hash index on it
  - **for each** tuple \(r\) in \(R\)
  - **use** the hash index on \(S\) to find tuples such that \(S.a = r.a\)

  - Cost: \(b_r + b_s\) transfers, 2 seeks (unchanged)
  - Why good?
    - CPU cost is much better (even though we don’t care about it too much)
    - Much better than nested-loops join when \(S\) doesn’t fit in memory (next)

Hash Join

- **Case 2:** Smaller relation \((S)\) doesn’t fit in memory

  Basic idea:
  
  - partition tuples of each relation into sets that have same value on join attributes
  - must be equi-/natural join

  Phase 1:
  
  - Read \(R\) block by block and partition it using a hash function: \(h1(a)\)
    - Create one partition for each possible value of \(h1(a)\) (\(n_h\) partitions)
  - Write the partitions to disk
    - \(R\) gets partitioned into \(R_1, R_2, \ldots, R_k\)
  - Similarly, read and partition \(S\), and write partitions \(S_1, S_2, \ldots, S_k\) to disk
  - Only requirements:
    - Room for a single input block and one output block for each hash value
    - Each \(S\) partition fits in memory
Hash Join

- Case 2: Smaller relation \((S)\) doesn't fit in memory
- Two “phases”
- Phase 2:
  - Read \(S_i\) into memory, and build a hash index on it \((S_i\) fits in memory\)
    - *Use a different hash function from the partition hash: \(h_2(a)\)*
  - Read \(R_i\) block by block, and use the hash index to find matches.
  - Repeat for all \(i\).
Hash Join

- **Case 2: Smaller relation \((S)\) doesn’t fit in memory**
- Two “phases”:
  - **Phase 1:**
    - Partition the relations using one hash function, \(h_1(a)\)
  - **Phase 2:**
    - Read \(S_i\) into memory, and build a hash index on it (\(S_i\) fits in memory)
    - Read \(R_i\) block by block, and use the hash index to find matches.
- **Cost**?
  - \(3(b_r + b_s)\) block transfers
    - \(R\) or \(S\) might have partially full block to be read and written \((\text{ignored})\)
  - \(+ 2\left(\frac{b_r}{b_b} + \frac{b_s}{b_b}\right)\) seeks \((\text{seek count unclear})\)
    - Where \(b_b\) is the size of each input buffer \((p\ 560)\)
    - Much better than Nested-loops join under the same conditions

Hash Join: Issues

- **How to guarantee that each partition of \(S\) fits in memory?**
  - Say \(S = 10,000\) blocks, Memory = \(M = 100\) blocks
  - Use a hash function that hashes to \(n_h=100\) different values?
    - Eg. \(h_1(a) = a \mod 100\)?
  - Problem: Impossible to guarantee uniform split
    - Some partitions will be larger than 100 blocks, some will be smaller
  - Use \(n_h=100*f\) different values
    - \(f\) is called **fudge factor**, typically around 1.2
    - So we may consider \(h_1(a) = a \mod 120.\)
    - This is okay IF \(a\) is nearly uniformly distributed
  - **Why can’t we just set \(h_n\) to 200?**
    - need to have a per-value output block in mem during build phase
Hash Join: Issues

Memory required?

- Say $S = 5000$ blocks, $M = 100$ blocks
  - So $50f = 60$ different partitions
  - During phase 1:
    - Need 1 block for storing $R$
    - Need 60 blocks for storing each partition of $R$
  - So will use 61 blocks of memory, all good

- What if $S = 10000$ blocks?
  - So $100f = 120$ different partitions
  - During phase 1:
    - Need 1 block for storing $R$
    - Need 120 blocks for storing each partition of $R$
  - So must have at least 121 blocks of memory
  - We only have 100 blocks
  - need recursive partitioning

Joins: Summary

- Block Nested-loops join
  - Can always be applied irrespective of the join condition
- Index Nested-loops join
  - Only applies if an appropriate index exists, good if not made index lookups
- Hash joins – only for equi-joins
  - Join algorithm of choice when the relations are large
- Sort-merge join
  - Very commonly used – especially since relations are typically sorted
  - Sorted results commonly desired at the output
    - To answer group by queries, for duplicate elimination, because of ASC/DSC
Query Processing

- Overview
- Selection operation
- Sorting
- Join operators
- Other operators
- Putting it all together…

Group By and Aggregation

```sql
select a, count(b)
from R
group by a;
```

- Hash-based algorithm:
  - Create a hash table on `a`, and keep the `count(b)` so far
  - Read `R` tuples one by one
  - For a new `R` tuple, “r”
    - Check if `r.a` exists in the hash table
    - If yes, increment the count
    - If not, insert a new value
Group By and Aggregation

```sql
select a, count(b)
from R
group by a;
```

- **Sort-based algorithm:**
  - Sort $R$ on $a$
  - Now all tuples in a single group are contiguous
  - Read tuples of $R$ (sorted) one by one and compute the aggregates

---

**Summary:**

- `sum()`, `count()`, `min()`, `max()`:
  - Only need to maintain one value per group
  - Called “distributive”
- `average()`:
  - Need to maintain the “sum” and “count” per group
  - Called “algebraic”
- `stddev()`:
  - Algebraic, but need to maintain more state
- `median()`:
  - Efficient with sort, but need two passes (called “holistic”)
  - Find num of tuples in each group, and then median tuple in each group
- `count(distinct b)`:
  - Must do duplicate elimination before the count
Duplicate Elimination

\[ \text{select distinct } a \]
\[ \text{from } R ; \]

- Best done using sorting – (hashing also works)
- Steps:
  - Sort the relation \( R \)
  - Read tuples of \( R \) in sorted order
  - \( \text{prev} = \text{null} ; \)
  - for each tuple \( r \) in \( R \) \( \text{(sorted)} \)
    - if \( r \neq \text{prev} \) then
      - Output \( r \)
      - \( \text{prev} = r \)
    - else
      - Skip \( r \)

Set operations

\((\text{select * from } R) \text{ union (select * from } S) ;\)
\((\text{select * from } R) \text{ intersect (select * from } S) ;\)
\((\text{select * from } R) \text{ union all (select * from } S) ;\)
\((\text{select * from } R) \text{ intersect all (select * from } S) ;\)

- Remember the rules about duplicates
- “union all”: just append the tuples of \( R \) and \( S \)
- “union”: append tuples of \( R \) and \( S \), then duplicate elimination
- “intersection”: similar to joins
  - Find tuples of \( R \) and \( S \) that are identical on all attributes
  - Can use hash-based or sort-based algorithm
Query Processing

- Overview
- Selection operation
- Sorting
- Join operators
- Other operators
- Putting it all together…

Evaluation of Expressions

```
select customer-name 
from account a, customer c 
where a.SSN = c.SSN and 
a.balance < 2500
```

- Two options:
  - Materialization
  - Pipelining
Evaluation of Expressions

- **Materialization**
  - Evaluate each expression separately
    - Store its result on disk in temporary relations
    - Read it for next operation

- **Pipelining**
  - Evaluate multiple operators simultaneously
    - Do not go to disk
    - Usually faster, but requires more memory
    - Also not always possible..
      - E.g. Sort-Merge Join
    - Harder to reason about

Materialization

- **Materialized evaluation always works**
- **Can be expensive to write and read back from disk**
  - Cost formulas ignore cost of writing final results to disk, so
    - Overall cost = Sum of costs of individual operations +
      - cost of writing intermediate results to disk

- **Double buffering:**
  - Use two output buffers for each operation, when one is full write it to disk, while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time
Pipelining

- Evaluate several operations at same time passing results from one to the next.
- E.g., in previous expression tree, don’t store result of $\sigma_{\text{balance}<2500}(\text{account})$
  - Instead, pass tuples directly to the join.
  - Similarly, don’t store result of join, pass tuples directly to projection.
- Much cheaper: no need to store a temporary relation to disk.
- Requires more memory
  - All operations are executing at the same time (say as processes)
- Somewhat limited applicability
- Beware blocking operations:
  - must consume entire input before it starts producing output tuples

Pipelining

- Operators must generate output tuples while reading inputs
  - Selection: Usually yes.
  - Sort: NO. The sort operation is blocking
  - Sort-merge join: The final (merge) phase can be pipelined
  - Hash join: The partitioning phase is blocking; the second phase can be pipelined
  - Aggregates: Typically no.
  - Duplicate elimination: Since it requires sort, the final merge phase could be pipelined
  - Set operations: see duplicate elimination
Pipelining: Demand-driven

- **Iterator Interface**
  - Each operator implements:
    - `init()`: Initialize the state (sometimes called `open()`)
    - `get_next()`: get the next tuple from the operator
    - `close()`: Finish and clean up
  - Example: sequential scan:
    - `init()`: open the file
    - `get_next()`: get the next tuple from file
    - `close()`: close the file
  - Execute by repeatedly calling `get_next()` at the root
    - root calls `get_next()` on its children, the children call `get_next()` on their children etc…
  - The operators need to maintain internal state so they know what to do when the parent calls `get_next()`

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()`)
  - 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 the PostgreSQL codebase (or assignment 7)

Pipelining (Cont.)

- In producer-driven or eager pipelining
  - Operators produce tuples eagerly and pass them up to their 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, and 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
- 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

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 relation = 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 (hash, vs merge, vs NL)
  - What join order?
    - Given a join on R, S, T, should join R with S first, or S with T?

- 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

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, $\sigma_{balance<2500}(account)$
    • 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
  - Optimizers 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

![Histogram Diagram]

**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](histogram.png)

Histories

- 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 \( \prod_A(r) \).
  - \( \text{MAX}(A, r) \): 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:

\[
b_r = \left\lfloor \frac{n_r}{f_r} \right\rfloor
\]
Selection Size Estimation

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

- $\sigma_{A \geq X}(r)$ (case of $\sigma_{A \leq X}(r)$ is symmetric)
  - Let $c$ be 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($\theta_i$) = the probability that a particular 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: $\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n}(r)$. 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:
  $$n_r \cdot \left(1 - \frac{S_1}{n_r}\right) \cdot \left(1 - \frac{S_2}{n_r}\right) \cdot \ldots \cdot \left(1 - \frac{S_n}{n_r}\right)$$

- negation: $\sigma_{\lnot\theta_i}(r)$. Estimated number of tuples: $n_r - size(\sigma_{\theta_i}(r))$
Estimating Output Sizes: Joins

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

- **CASE 1:** \( a \) is key for \( S \)
  - *Worst case: each tuple of \( R \) joins with exactly one tuple of \( S \)*
  - So: \(|R \ JOIN \ S| = |R| = 10,000\)

- **CASE 2:** \( a \) is key for \( R \)
  - Each \( S \) tuple can match w/ only a single \( R \) tuple.
  - So: \(|R \ JOIN \ S| = |S| = 5,000\)

Equijoins simplify things.

---

Estimating Output Sizes: Joins

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

- **CASE 3:** \( a \) is not a key for either
  - Reason with the distributions on \( a \)
  - Say: the domain of \( a: V(A, R) = V(A, S) = 100\)
  - 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 \( a \)
      - So each distinct value will produce \( 100 \times 50 = 5000 \) tuples
        - this assumes that the distinct values in \( R,S \) are the same
      - So total number of results in the join:
        - \( 5000 / \text{distinctValue} \times 100 \) distinct values = \( 500,000 \) total tuples
    - We can improve the accuracy if we know the distributions on \( a \) better
      - Say using a histogram