Query Processing

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

Join

- \[ \text{select } * \text{ from } R, S \text{ where } R.a = S.a \]
  - Called an “equi-join”
- \[ \text{select } * \text{ from } R, S \text{ where } |R.a - S.a| < 0.5 \]
  - Not an “equi-join”

- Option 1: Nested-loops
  
  \[ \text{for each tuple } r \text{ in } R \]
  
  \[ \text{for each tuple } s \text{ in } S \]
  
  \[ \text{check if } r.a = s.a \text{ (or whether } |r.a - s.a| < 0.5) \]

- Can be used for any join condition
  - As opposed to some algorithms we will see later
- R called outer relation
- S called inner relation
Nested-loops Join

- Cost? Depends on the actual values of parameters, especially memory
- \( b_r, b_s \to Number \ of \ blocks \ of \ R \ and \ S \)
- \( n_r, n_s \to Number \ of \ tuples \ of \ R \ and \ S \)
- **Case 1:** Minimum memory required = 3 blocks
  - One to hold the current \( R \) block, one for current \( S \) block, one for the result being produced
  - Blocks transferred:
    - Must scan \( R \) tuples once: \( b_r \)
    - For each \( R \) tuple, must scan \( S \): \( n_r \times b_s \)
  - Seeks?
    - \( n_r + b_r \)

---

**Case 1: Minimum memory required = 3 blocks**

- Blocks transferred: \( n_r \times b_s + b_r \)
- Seeks: \( n_r + b_r \)

**Example:**
- Number of records -- \( R \): \( n_r = 10,000 \), \( S \): \( n_s = 5000 \)
- Number of blocks -- \( R \): \( b_r = 400 \), \( S \): \( b_s = 100 \)

**\( R \) "outer relation":**
- blocks transferred: \( n_r \times b_s + b_r = 10000 \times 100 + 400 = 1,000,400 \)
- seeks: 10400
- time: \( 1000400 \ t_r + 10400 \ t_S = 1000400(\text{.1ms}) + 10400(4\text{ms}) = 1020.8 \text{ sec} \)

**\( S \) outer relation?**
- \( 5000 \times 400 + 100 = 2,000,100 \) block transfers,
- 5100 seeks
- \( = 2000100 \ t_r + 5100 \ t_S = 2041.7\text{ sec} \)

*Order matters!*
Nested-loops Join

- **Case 2: S fits in memory**
  - Blocks transferred: $b_s + b_r$
  - Seeks: 2

- **Example**:
  - Number of records -- $R$: $n_r = 10,000$, $S$: $n_s = 5000$
  - Number of blocks -- $R$: $b_r = 400$, $S$: $b_s = 100$

- **Then**:
  - blocks transferred: $400 + 100 = 500$
  - seeks: 2
  - $= 500t_T + 2t_S = 0.058$sec

*Orders of magnitude difference*

Block Nested-loops Join

- **Simple modification to “nested-loops join”** (block at a time)
  
  for each block $B_r$ in $R$
  
  for each block $B_s$ in $S$
  
  for each tuple $r$ in $B_r$
  
  for each tuple $s$ in $B_s$
  
  check if $r.a = s.a$ (or whether $|r.a - s.a| < 0.5$)

- **Case 1: Minimum memory required = 3 blocks**
  - Blocks transferred: $b_r * b_s + b_r$
  - Seeks: $2 * b_r$

- **For the example**:
  - blocks: 40400, seeks: $800 = 4.04 + 3.2 = 7.24$ sec
Block Nested-loops Join

- **Case 1**: Minimum memory required = 3 blocks
  - Blocks transferred: $b_r \cdot b_s + b_r$
  - Seeks: $2 \cdot b_r$

- **Case 2**: S fits in memory
  - Blocks transferred: $b_s + b_r$
  - Seeks: 2

- **What about in between?**
  - Say there are 50 blocks, but $S$ is 100 blocks
  - Why not use all the memory that we can…

- **Case 3**: 50 blocks ($S = 100$ blocks)
  
  *for each group of 48 blocks in $R*$
  
  *for each block $B_s$ in $S$*
  
  *for each tuple $r$ in the group of 48 blocks*
  
  *for each tuple $s$ in $B_s* $

  check if $r.a = s.a$ (or whether $|r.a - s.a| < 0.5$)

- Why is this good?
  - We only have to read $S$ a total of $b_r/48$ times (instead of $b_r$ times)
  - Blocks transferred: $b_s \cdot b_r/48 + b_r = 100 \cdot 400/48 + 400 = 1233$
  - $b_s \cdot b_r/48 + b_r = 400 \cdot 100/48 + 100 = 933$ (but more seeks)
  - Seeks: $2 \cdot b_r/48$
Index Nested-loops Join

- `select * from R, S where R.a = S.a`
  - “equi-join”
- Nested-loops
  - `for each tuple r in R`
    - `for each tuple s in S`
      - `check if r.a = s.a (or whether |r.a - s.a| < 0.5)`
- Suppose there is an index on `S.a`
- Why not use the index instead of the inner loop?
  - `for each tuple r in R`
    - `use the index to find S tuples with S.a = r.a`

Cost of the join:
- \( b_r(t_r + t_s) + n_r \times c \)
- \( c == \text{the cost of index access} \)
  - `Computed using the formulas discussed earlier`
Index Nested-loops Join

- With indexes for both $R$, $S$, use one with fewer tuples as outer.
- Recall example:
  - Number of records -- $R$: $n_r = 10,000$, $S$: $n_s = 5000$
  - Number of blocks -- $R$: $b_r = 400$, $S$: $b_s = 100$
- Assume $B^+$-tree for $R$, avg fanout of 20, implies height $R$ is 4
  - Cost is $100 + 5000 \times (4 + 1) = 25,100$, each with seek and transfer
- Assume $B^+$-tree is on $S$: height = 3
  - Cost is $400 + 10000 \times (3+1) = 40,400$, each with seek and transfer

Index Nested-loops Join

- Restricted applicability
  - An appropriate index must exist
  - What about $|R.a - S.a| < 5$?
- Great for queries with joins and selections

```
SELECT *
FROM accounts, customers
WHERE accounts.customer-SSN = customers.customer-SSN AND
  accounts.acct-number = "A-101"
```
- Use `accounts` as outer, use select to prune reads of customers
So far…

- **Block Nested-loops join**
  - Can always be applied irrespective of the join condition
  - If the smaller relation fits in memory, then cost:
    - \( b_r + b_s \)
    - This is the best we can hope if we have to read the relations once each
  - CPU cost of the inner loop is high
  - Typically used when the smaller relation is really small (few tuples) and index nested-loops can’t be used

- **Index Nested-loops join**
  - Only applies if an appropriate index exists
  - Very useful when we have selections that return small number of tuples
    - `select balance from customer, accounts where customer.name = "j. s." and customer.SSN = accounts.SSN`

---

Recall: External Sorting Using Sort-Merge (N \( \geq \) M)

<table>
<thead>
<tr>
<th>Initial relation</th>
<th>Create runs</th>
<th>Merge pass-1</th>
<th>Merge pass-2</th>
<th>Sorted output</th>
</tr>
</thead>
<tbody>
<tr>
<td>g 24</td>
<td>a 19</td>
<td>a 19</td>
<td>a 14</td>
<td></td>
</tr>
<tr>
<td>a 24</td>
<td>d 31</td>
<td>b 14</td>
<td>d 31</td>
<td></td>
</tr>
<tr>
<td>d 31</td>
<td>c 33</td>
<td>c 33</td>
<td>e 16</td>
<td></td>
</tr>
<tr>
<td>c 33</td>
<td>b 14</td>
<td>d 31</td>
<td>g 24</td>
<td></td>
</tr>
<tr>
<td>b 14</td>
<td>e 16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e 16</td>
<td>d 21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r 16</td>
<td>m 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d 21</td>
<td>p 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m 3</td>
<td>d 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p 2</td>
<td>a 14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g 24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( M = 3 \)

\( N = 12 \)

\( b_r (2 \lceil \log_{b_r} (b_s / M) \rceil + 1) \) blocks

Seeks:

\[ 2 \lceil b_s / M \rceil + \lceil b_s / b_r \rceil (2 \lceil \log_{b_s} (b_r / M) \rceil - 1) \]
Merge-Join (Sort-merge join)

- Pre-condition:
  - equi-/natural joins
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this
- Called “sort-merge join” sometimes

```sql
SELECT *
FROM r, s
WHERE r.a1 = s.a1
```

Step:
1. Compare the tuples at pr and ps
2. Move pointers down the list
   - Depending on the join condition
3. Repeat

![Diagram of Merge-Join]

Merge-Join (Sort-merge join)

- Cost:
  - If the relations sorted, then just
    - $b_r + b_s$ block transfers, some seeks depending on memory size
  - What if not sorted?
    - Then sort the relations first
    - In many cases, still very good performance
    - Typically comparable to hash join
- Observation:
  - The final join result will also be sorted on $a1$
  - This might make further operations easier to do
    - E.g. duplicate elimination
So far...

- **Block Nested-loops join**
  - Can always be applied irrespective of the join condition

- **Index Nested-loops join**
  - Only applies if an appropriate index exists
  - Very useful when we have selections that return small number of tuples
    - select balance from customer, accounts where customer.name = "j. s." and customer.SSN = accounts.SSN

- **Merge joins**
  - Join algorithm of choice when the relations are large
  - Sorted results commonly desired at the output
    - To answer group by queries, for duplicate elimination, because of ASC/DSC

---

**Hash Join**

- **Case 1: Smaller relation \(S\) fits in memory**
- Nested-loops join:
  
  ```
  for each tuple \(r\) in \(R\)
  for each tuple \(s\) in \(S\)
  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:
  
  ```
  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 \(S.a = r.a\)
  ```
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: \(h_1(a)\)
      - Create one partition for each possible value of \(h_1(a)\) \((n_r\) 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 \(i\).

\(n_h = 5\) num hash values
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 (ignored)
  - \( + 2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) \) seeks (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 100 different values?
    - Eg. \( h_1(a) = a \% 100 \)?
  - Problem: Impossible to guarantee uniform split
    - Some partitions will be larger than 100 blocks, some will be smaller
  - Use a hash function that hashes to \( 100*f \) different values
    - \( f \) is called fudge factor, typically around 1.2
    - So we may consider \( h_1(a) = a \% 120. \)
    - 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 $S = 10000$ blocks, $Memory = M = 100$ blocks
  - So 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
- **Typically need $\sqrt{|S| \times f}$ blocks of memory**
  - So if $S$ is 10000 blocks, and $f = 1.2$, need 110 blocks of memory
  - Need:
    - $M > n_h + 1$
  - each partition of $S$ to fit in $M-1$ (why not $R$?)
  - space for hash build on $h2()$ (usually ignored)
- Example:
  - $h_h = 109$, average size = $10,000/109 = 91.7$

Hash Join: If $S_i$ Too Large

- **Avoidance**
  - Fudge factor
- **Resolution**
  - partition w/ a third hash $h3()$
  - also partition $R_i$
  - go through each sub-partition
  - this approach could be used for *every* partition
Merge-Join (Sort-merge join)

- Pre-condition:
  - equi-/natural joins
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this
- Called “sort-merge join” sometimes

\[
\text{select } * \\
\text{from } r, s \\
\text{where } r.a1 = s.a1
\]

Step:
1. Compare the tuples at \( p_r \) and \( p_s \)
2. Move pointers down the list
   - Depending on the join condition
3. Repeat

Merge-Join (Sort-merge join)

- Cost:
  - If the relations sorted, then just
    - \( b_r + b_s \) block transfers, some seeks depending on memory size
  - What if not sorted?
    - Then sort the relations first
    - In many cases, still very good performance
    - Typically comparable to hash join
- Observation:
  - The final join result will also be sorted on \( a1 \)
  - This might make further operations easier to do
    - E.g. duplicate elimination
Query Processing

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

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
- 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

```
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

\[
\text{select } a, \text{ count}(b) \\
\text{from } R \\
group by a;
\]

- 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:**
- \( \text{sum()} \), \( \text{count()} \), \( \text{min()} \), \( \text{max()} \): only need to maintain one value per group
  - Called “distributive”
- \( \text{average()} \): need to maintain the “sum” and “count” per group
  - Called “algebraic”
- \( \text{stddev()} \): algebraic, but need to maintain some more state
- \( \text{median()} \): can do efficiently with sort, but need two passes (called “holistic”)
  - First to find the number of tuples in each group, and then to find the median tuple in each group
- \( \text{count}(\text{distinct } b) \): must do duplicate elimination before the count
Duplicate Elimination

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

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

Set operations

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

- Remember the rules about duplicates
- “union all”: just append the tuples of \( R \) and \( S \)
- “union”: append the tuples of \( R \) and \( S \), and \text{do duplicate elimination}
- “intersection”: similar to joins
- Find tuples of \( R \) and \( S \) that are identical on all attributes
- Can use \text{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

- Need operators that generate output tuples while receiving tuples from their 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 = 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 = 16ms
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

\[
\Pi_{\text{customer\_name}}
\]

\[\sigma_{\text{branch\_city}=\text{Brooklyn}}\]

\[
\Pi_{\text{customer\_name}}
\]

\[\sigma_{\text{branch\_city}=\text{Brooklyn}}\]

\[
\text{branch} \rightarrow \text{account} \rightarrow \text{depositor}
\]

\[
\text{branch} \rightarrow \text{account} \rightarrow \text{depositor}
\]
Query Optimization

- Execution plans
  - Evaluation expressions annotated with the methods used

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, \( \sigma_{\text{balance}<2500}(\text{account}) \)
    - 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 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

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: \( \frac{45,000}{80} = 562.50 \)
- More Information → Better estimation

![Histogram Chart]

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 $\Pi_A(r)$.
  - $\text{MAX}(A, r)$: the 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

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

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

- conjunction: $\sigma_{\theta_1 \land \theta_2 \land \ldots \land \theta_n}(r)$
  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:
  $$ n_r \times \left(1 - \frac{S_1}{n_r}\right) \times \left(1 - \frac{S_2}{n_r}\right) \times \ldots \times \left(1 - \frac{S_n}{n_r}\right) $$

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