Query Processing

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

Example: External Sorting Using Sort-Merge (N >= M)

M = 3
N = 12
External sorting cont.

seeks can be reduced by adding more blocks

<table>
<thead>
<tr>
<th>d</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>3</td>
</tr>
<tr>
<td>r</td>
<td>16</td>
</tr>
</tbody>
</table>

\[ h_2() \]

<table>
<thead>
<tr>
<th>a</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>7</td>
</tr>
<tr>
<td>p</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ h_3() \]

/ 2 in this example

Hash Join: If \( S_i \) Too Large

- **Avoidance**
  - Fudge factor

- **Resolution**
  - partition w/ a third hash \( h_3() \)
  - 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

```
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

Cost:
- If each set of S tuples w/ the same value for the join key fits into memory, and 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
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
- Join operators
- Other operators
- Putting it all together…
- Sorting
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

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
Group By and Aggregation

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

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

Duplicate Elimination

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

- 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

(select * from R) union (select * from S)
(select * from R) intersect (select * from S)
(select * from R) union all (select * from S)
(select * from R) intersect all (select * from S)

- Remember the rules about duplicates
- "union":
  - append the tuples of $R$ and $S$
  - duplicate elimination
- "union all":
  - just append the tuples of $R$ and $S$
- "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
- Join operators
- Other operators
- Putting it all together...
- Sorting
Evaluation of Expressions

Two options:
- Materialization
- Pipelining

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
  \( \sigma_{balance<2500}(account) \)
  - 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:
  - blocking operations consume entire input before producing any 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()
Example: 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

Example: 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 query_processing.py)
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