Query Processing

continued...

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

  - 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 \(h1(a)\)
    - 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)
  - Using 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

- $n_h = 5$
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 \( \text{ (ignored) } \)
  - \( + 2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \) 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 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 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 $\sqrt{|S| \cdot f}$ blocks of memory
  - So if $S$ is 10000 blocks, and $f = 1.2$, need 110 blocks of memory
  - Need:
    - $M > n_1 + 1$
    - each partition of $S$ to fit in $M-1$ (why not $R$?)
    - space for hash build on $h2()$ (usually ignored)

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

```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

---

**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
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

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

- **Hash-based algorithm:**
  - Create a hash table on \( a \), and keep the \( \text{count}(b) \) 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

- **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

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

- `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 some more state
- `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
- `count(distinct b)`: must do duplicate elimination before the count

Duplicate Elimination

```
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
  - `prev = null;`
  - for each tuple `r` in `R (sorted)`
    - if `r != prev` then
      - Output `r`
      - `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 all”: just append the tuples of R and S
- “union”: append the tuples of R and S, and do 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
- 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 is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing 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 $\text{balance} < 2500(\text{account})$
  - 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

- **close():**
  - Call `close()` on the left and the right children
  - Delete the hash table, other intermediate state etc…
- **get_next():** (*partitioning needed*)
  - **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
Pipelining (Cont.)

- In produce-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