Query Processing

continued...

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
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 $a_1$
  - This might make further operations easier to do
    - E.g. duplicate elimination

Query Processing

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

- Commonly required for many operations
  - Duplicate elimination, group by’s, sort-merge join
  - Queries may have ASC or DSC in the query
- One option:
  - Read the lowest level of B+-tree
    - May be enough in many cases
  - But if relation not sorted, too many random accesses
- If relation small enough…
  - Read in memory, use quicksort (qsort() in C)
- What if relation too large to fit in memory?
  - External sort-merge

External sort-merge

- Divide and Conquer !!
- Let $M$ denote the memory size (in blocks)
- Phase 1:
  - Read first $M$ blocks of relation, sort, and write it to disk
  - Read the next $M$ blocks, sort, and write to disk …
  - Say we have to do this “N” times
  - Result: $N$ sorted runs of size $M$ blocks each
- Phase 2:
  - Merge the $N$ runs ($N$-way merge)
  - Can do it in one shot if $N < M$
External sort-merge

- **Phase 1:**
  - Create *sorted runs of size* $M$ *each*
  - Result: $N$ sorted runs of size $M$ blocks each

- **Phase 2:**
  - Merge the $N$ runs (*$N$-way merge*)
  - Can do it in one shot if $N < M$

- **What if $N > M$?**
  - Do it recursively
  - Not expected to happen
  - If $M = 1000$, can compare 1000 runs
    - (4KB blocks): can sort: 1000 runs, each of 1000 blocks, each of 4k bytes = 4GB of data

---

Example: 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>a 19</td>
<td>a 19</td>
<td>a 14</td>
<td>a 14</td>
<td>M = 3</td>
</tr>
<tr>
<td>g 24</td>
<td>g 24</td>
<td>g 24</td>
<td>g 24</td>
<td>N = 12</td>
</tr>
<tr>
<td>d 31</td>
<td>d 31</td>
<td>d 31</td>
<td>d 31</td>
<td></td>
</tr>
<tr>
<td>c 33</td>
<td>c 33</td>
<td>c 33</td>
<td>c 33</td>
<td></td>
</tr>
<tr>
<td>b 14</td>
<td>b 14</td>
<td>b 14</td>
<td>b 14</td>
<td></td>
</tr>
<tr>
<td>e 16</td>
<td>e 16</td>
<td>e 16</td>
<td>e 16</td>
<td></td>
</tr>
<tr>
<td>r 16</td>
<td>r 16</td>
<td>r 16</td>
<td>r 16</td>
<td></td>
</tr>
<tr>
<td>d 21</td>
<td>d 21</td>
<td>d 21</td>
<td>d 21</td>
<td></td>
</tr>
<tr>
<td>m 3</td>
<td>m 3</td>
<td>m 3</td>
<td>m 3</td>
<td></td>
</tr>
<tr>
<td>p 2</td>
<td>p 2</td>
<td>p 2</td>
<td>p 2</td>
<td></td>
</tr>
<tr>
<td>d 7</td>
<td>d 7</td>
<td>d 7</td>
<td>d 7</td>
<td></td>
</tr>
<tr>
<td>a 14</td>
<td>a 14</td>
<td>a 14</td>
<td>a 14</td>
<td></td>
</tr>
</tbody>
</table>
External Merge Sort (Cont.)

- **Cost analysis:**
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
  - Disk for initial run creation as well as in each pass is $2b_r$
    - for final pass, we don’t count write cost
    - output may be pipelined (sent via memory to parent operation)

Thus total number of disk transfers for external sorting:

$$2b_r \lceil \log_{M-1}(b_r/M) \rceil - b_r + 2b_r = b_r (2 \lceil \log_{M-1}(b_r/M) \rceil + 1) = 12 \times (2 \times 2 + 1) = 60$$

Seeks:

$$2 \lceil b_r/M \rceil + 2b_r \lceil \log_{M-1}(b_r/M) \rceil - b_r = 2 \lceil b_r/M \rceil + b_r (2 \lceil \log_{M-1}(b_r/M) \rceil - 1) = 8 + 12(2 \times 2 - 1) = 44$$

#blocks read at a time, $b_o$, ignored here

---

External Merge Sort (Cont.)

- What if $M = 4$?
  - merge in one round (3 runs, 1 buffer for output).
  - disk transfers: $= b_r (2 \lceil \log_{M-1}(b_r/M) \rceil + 1) = 12 \times (2 \times 1 + 1) = 36$
  - seeks: $= 2 \lceil b_r/M \rceil + b_r (2 \lceil \log_{M-1}(b_r/M) \rceil - 1) = 6 + 12(2 - 1) = 18$
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)
    - 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

\[ n_h = 5 \]
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 \( S \) into memory, and build a hash index on it (\( S \) fits in memory)
    - Read \( R \) 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)
  - \( + 2\left( \lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil \right) \) 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. \( h1(a) = a \% 100 \)?
  - 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 \( 100f \) different values
    - \( f \) is called fudge factor, typically around 1.2
    - So we may consider \( h1(a) = a \% 120 \).
    - 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 $\sqrt{|S| \times f}$ blocks of memory
  - So if $S$ is 10000 blocks, and $f = 1.2$, need 110 blocks of memory
  - Need:
    - $M > n_s + 1$
    - each partition of $S$ to fit in $M-1$ (why not $R$?)
    - space for hash build on $h_2()$ (usually ignored)

- Example:
  - $h_n = 109$, average size = $10,000/109 = 91.7$

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**

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

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

**Cost:**
- If 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

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

- \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

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

- Remember the rules about duplicates
- “union all”:
  - just append the tuples of \(R\) and \(S\)
- “union”:
  - append the tuples of \(R\) and \(S\)
  - 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

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 
  \[ \sigma_{balance < 2500}(account) \]
  - 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:
  - 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()