Query Processing

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

Join

- `select * from R, S where R.a = S.a`
  - Called an "equi-join"
- `select * from R, S where |R.a – S.a| < 0.5`
  - Not an "equi-join"

- Option 1: 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)`
  - 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
- \( b_r, b_s \rightarrow \text{Number of blocks of } R \text{ and } S \)
- \( n_r, n_s \rightarrow \text{Number of tuples of } R \text{ and } S \)
- **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: \( 1000400 \times t_r + 10400 \times t_s = 1000400(0.1\text{ms}) + 10400(4\text{ms}) = 1020.8 \text{ sec} \)
- \( S \) "outer relation?"
  - \( 5000 \times 400 + 100 = 2,000,100 \) block transfers,
  - 5100 seeks
  - \( = 2000100 \times t_r + 5100 \times t_s = 2041.7 \text{ sec} \)

*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
  - $= 500t_r + 2t_s = 0.058$ 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: \( b_r \times b_s + b_r \)
  - Seeks: \( 2 \times b_r \)

- **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: \( b_s \times b_r/48 + b_r = 100 \times 400/48 + 400 = 1233 \)
  - Or \( b_s \times b_r/48 + b_r = 400 \times 100/48 + 100 = 933 \) (but more seeks)
  - Seeks: \( 2 \times b_r/48 \)
Index Nested-loops Join

- \textit{select * from R, S where R.a = S.a} \\
  - “equi-join” \\
- Nested-loops
  
  \textit{for each tuple r in R} \\
  \hspace{1em} \textit{for each tuple s in S} \\
  \hspace{2em} \textit{check if r.a = s.a (or whether |r.a – s.a| < 0.5)} \\

- Suppose there is an index on \textit{S.a} \\
- \textit{Why not use the index instead of the inner loop?} \\
  \textit{for each tuple r in R} \\
  \hspace{1em} \textit{use the index to find S tuples with S.a = r.a} \\

Cost of the join:

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

- W/ indexes for both \( R, S \), use one w/ 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 w/ seek and transfer
  - Assume B+-tree is on \( S \): height = 3
    - Cost is \( 400 + 10000 \times (3+1) = 40,400 \), each w/ seek and transfer

- Restricted applicability
  - An appropriate index must exist
  - What about \( |R.a - S.a| < 5 \)?
- Great for queries with joins and selections
  
  ```sql
  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 >= M)

![Sorting Diagram]

\[ b_r \left( 2 \left[ \log_{M'}(b_r/M) \right] + 1 \right) \text{ blocks} \]

\[ 2 \left[ \frac{b_r}{M} \right] + \left[ \frac{b_r}{b_s} \right] \left( 2 \left[ \log_{M'}(b_r/M) \right] - 1 \right) \text{ seeks} \]
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

![Diagram](image)

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: $h1(a)$
      - Create one partition for each possible value of $h1(a)$ ($n_r$ 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 \( i \).
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)
  - \(+ 2\left(\left\lceil b_r/b_b\right\rceil + \left\lceil b_s/b_b\right\rceil\right)\) 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. \(h_1(a) = a \mod 100\) ?
  - Problem: Impossible to guarantee uniform split
    - Some partitions will be larger than 100 blocks, some will be smaller
  - Use a hash function that hashes to \(100*f\) different values
    - \(f\) is called fudge factor, typically around 1.2
    - So we may consider \(h_1(a) = a \mod 120\).
    - 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 $S = 10000$ 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:
    - $M > n_r + 1$
    - each partition of $S$ to fit in $M-1$ (why not $R$?)
    - space for hash build on $h2()$ (usually ignored)
  - Example:
    - $h_r = 109$, 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**

```
select *
from r, s
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

**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