Query Processing

- Overview
- Selection operation
- Sorting
- Join operators
- 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 $N < M$

- What if $N \geq M$?
  - Do it recursively
  - If $M = 1000$, can only merge 999 runs
Example: External Sorting Using Sort-Merge \((b_r \geq M)\)

\[ M = 3 \]
\[ b_r = 12 \]

\(\text{blocksize} == \text{tuplesize for this example, not in general}\)

External Merge Sort (Cont.)

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

Thus total number of disk transfers for external sorting:
\[ b_r \times (2 [\log_{M-1}(b_r/M)] + 1) \]

Seeks:
\[ 2 \lfloor b_r / M \rfloor + \lfloor b_r / b_b \rfloor \times (2 [\log_{M-1}(b_r/M)] - 1) \]

\(b_b\) is:
- #blocks read at a time, and how many output blocks needed
- Assumed to be “1” unless otherwise specified
Example: External Sorting Using Sort-Merge ($b_r >= M$)

```
<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>M = 3</td>
</tr>
<tr>
<td>a 24</td>
<td>d 31</td>
<td>b 14</td>
<td>e 16</td>
<td>b_r = 12</td>
</tr>
<tr>
<td>d 31</td>
<td>c 33</td>
<td>g 24</td>
<td>d 31</td>
<td></td>
</tr>
<tr>
<td>c 33</td>
<td>b 14</td>
<td>e 16</td>
<td>d 7</td>
<td></td>
</tr>
<tr>
<td>b 14</td>
<td>e 16</td>
<td>g 24</td>
<td>d 21</td>
<td></td>
</tr>
<tr>
<td>e 16</td>
<td>r 16</td>
<td>d 7</td>
<td>g 24</td>
<td></td>
</tr>
<tr>
<td>r 16</td>
<td>d 21</td>
<td>m 3</td>
<td>m 3</td>
<td></td>
</tr>
<tr>
<td>d 21</td>
<td>m 3</td>
<td>r 16</td>
<td>p 2</td>
<td></td>
</tr>
<tr>
<td>m 3</td>
<td>p 2</td>
<td>r 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p 2</td>
<td>d 7</td>
<td>a 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d 7</td>
<td>p 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

block transfers:
$$b_r \cdot \left( \frac{2^{\lceil \log_{M^{-1}}(b_r / M) \rceil}}{b_r} \right) + 1 \right) = 60$$

seeks:
$$2 \cdot \left[ b_r / M \right] + \left[ b_r / b_b \right] \cdot \left( \lfloor \log_{M^{-1}}(b_r / M) \rfloor - 1 \right) = 44$$

$b_b$ for reading blocks at a time

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
- $b_r, b_s \rightarrow$ Number of blocks of $R$ and $S$
- $n_r, n_s \rightarrow$ 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?
    - $n_r + b_r$
Nested-loops Join

- **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 -- \( R: n_r = 10,000, S: n_s = 5000 \) \( t_r = 0.1ms \)
  - Number of blocks -- \( R: b_r = 400, S: b_s = 100 \) \( t_s = 4ms \)
  - \( R \) "outer relation":
    - Blocks transferred: \( n_r \times b_s + b_r = 10000 \times 100 + 400 = 1,000,400 \)
    - Seeks: 10400
    - Time: \( 1000400 t_r + 10400 t_s = 1000400(0.1ms) + 10400(4ms) = 141.64 \text{ sec} \)
- **S outer relation?**
  - \( 5000 \times 400 + 100 = 2,000,100 \) block transfers,
  - 5100 seeks
  - \( = 2000100 t_r + 5100 t_s = 220.41 \text{ sec} \)

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
  - \( = 500 t_r + 2 t_s = 0.058 \text{ sec} \)

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: $b_r \ast b_s + b_r$
  - Seeks: $2 \ast b_r$

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

- **Case 2: $S$ fits in memory**
  - Blocks transferred: $b_s + b_r$
  - Seeks: 2
  - Cost: $(500) t_r + 2t_s = 58$ msec

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

- **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: \( b_s * b_r / 48 + b_r = 100*400/48 + 400 = 1233 \)
    - Or \( b_s * b_r / 48 + b_r = 400*100/48 + 100 = 933 \) (but more seeks)
  - Seeks: \( 2 * b_r / 48 \)
  - Cost: \( 1233t_r + [800/48] t_s = 1233 + 17*4 = 1.3 \) sec

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 \( S.a = r.a \)
Index Nested-loops Join

- `select * from R, S where R.a = S.a`
  - Called an “equi-join”
  - 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:

  - \( b_r + n_r \times c_s \) (seeks and block transfers)

  - \( c_s \) is cost of index access for S
    - 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 -- \( R: b_r = 400, S: b_s = 100 \)

  ```
  b_{out} + n_{out} \times c_{in}
  ```

- 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
Index Nested-loops Join

- **Restricted applicability**
  - An appropriate index must exist
  - What about $|R.a - S.a| < 5$?  
    
    nope

- **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
    
    ```sql
    select balance from customer, accounts where customer.name = "j. s." and customer.SSN = accounts.SSN
    ```
Recall: External Sorting Using Sort-Merge \((b_r \geq M)\)

\[ M = 3 \]
\[ b_r = 12 \]

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

\[
2 \left\lfloor b_r / M \right\rfloor + \left\lfloor b_r / b_r \right\rfloor \left(2 \left\lfloor \log_{M}(b_r / M) \right\rfloor - 1 \right)
\]

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
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