Query Processing
Basic Steps in Query Processing

1. Query
2. Parser and Translator
3. Relational Algebra Expression
4. Optimizer
5. Execution Plan
6. Evaluation Engine
7. Data
8. Statistics about Data
9. Query Output
SELECT balance
FROM account
WHERE balance < 2500
SELECT balance
FROM account
WHERE balance < 2500
Measures of Query Cost

- Many factors contribute to time cost
  - disk accesses, CPU, or even network communication
- Disk access is the predominant
- Costs depends on the size of the buffer in main memory
Point Query

SELECT * 
FROM account 
WHERE balance = 2500

A1 (linear search) 

Let $br$ denotes number of blocks containing records from relation $r$

Cost = $b_{account}$

If selection is on a key attribute, average cost = ($b_{account}$/2)
Point Query

SELECT * 
FROM account 
WHERE balance = 2500

Cost = $\left\lceil \log_2(b_{account}) \right\rceil + b$ records that satisfy selection condition

If selection is on a key attribute, avg cost = $\left\lceil \log_2(b_{account}) \right\rceil$
Point Query Using Indices

**SELECT** *
FROM account
WHERE balance = 2500

A3 *(primary index on candidate key, equality)*

Cost = \( HT_i + 1 \)

If retrieve multiple records.
- Records will be on consecutive blocks
- \( Cost = HT_i + \text{number of blocks containing retrieved records} \)

1/14/2005 Yan Huang - CSCI5330 Database Implementation – Query Processing
Point Query Using Indices

SELECT *
FROM account
WHERE balance = 2500

Cost = HT_i + 1

If retrieve multiple records.
  - Records may not be consecutive
  - Cost = HT_i + number of records retrieved
Simple Range Query

```
SELECT *
FROM account
WHERE balance > 2500
```

- a linear file scan
  - Cost = $b_{account}$

- binary search
  - $\text{Cost} = \lceil \log_2(b_{account}) \rceil + b \text{ records that satisfy selection condition}$
Simple Range Query Using Indices

SELECT *
FROM account
WHERE balance > 2500

A6 (primary index)

Cost = $HT_i + \text{number of blocks containing retrieved records}$
Simple Range Query Using Indices

SELECT * 
FROM account
WHERE balance > 2500

Cost = $HT_i + \text{number of records retrieved}$
Simple Range Query

- What about:
  
  ```sql
  SELECT *
  FROM account
  WHERE balance < 2500
  ```
  
  - and
  
  ```sql
  SELECT *
  FROM account
  WHERE balance between 1500 and 2500
  ```
Implementation of Complex Selections

- **Conjunction:** \( \sigma_{\theta_1 \land \theta_2 \land \ldots \land \theta_n}(r) \)

- **A8 (one index).**
  - Select a combination of \( \theta_i \) and algorithms A1 through A7 that results in the least cost for \( \sigma_{\theta_i}(r) \).
  - Test other conditions on tuple after fetching it into memory buffer

- **A9 (multiple-key index).**
  - Use appropriate composite (multiple-key) index if available.

- **A10 (intersection of identifiers).**
  - Use corresponding index for each condition
  - Take intersection of all the obtained sets of record pointers.
Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n}(r)$

- **A11 (union of identifiers)**
  - Applicable if all conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.

- **Negation:** $\sigma_{-\theta}(r)$
  - Use linear scan on file
  - If very few records satisfy $-\theta$, and an index is applicable to $\theta$
We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple.

For relations that fit in memory, techniques like quicksort can be used. For relations that don’t fit in memory, **external sort-merge** is a good choice.
Example: External Sorting Using Sort-Merge

M=3

<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>a 14</td>
</tr>
<tr>
<td>a 24</td>
<td>d 31</td>
<td>b 14</td>
<td>a 19</td>
<td>a 19</td>
</tr>
<tr>
<td>d 31</td>
<td>g 24</td>
<td>c 33</td>
<td>a 19</td>
<td>a 19</td>
</tr>
<tr>
<td>c 33</td>
<td>b 14</td>
<td>d 31</td>
<td>b 14</td>
<td>b 14</td>
</tr>
<tr>
<td>b 14</td>
<td>c 33</td>
<td>e 16</td>
<td>c 33</td>
<td>c 33</td>
</tr>
<tr>
<td>e 16</td>
<td>d 21</td>
<td>g 24</td>
<td>d 21</td>
<td>d 21</td>
</tr>
<tr>
<td>r 16</td>
<td>d 21</td>
<td></td>
<td>d 21</td>
<td></td>
</tr>
<tr>
<td>d 21</td>
<td>m 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m 3</td>
<td>r 16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p 2</td>
<td>a 14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d 7</td>
<td>d 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a 14</td>
<td>p 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r 16</td>
<td>r 16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
External Merge Sort

Cost analysis:

- Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
- Disk accesses for initial run creation as well as in each pass is $2b_r$
  - for final pass, we don’t count write cost
    - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk

Thus total number of disk accesses for external sorting:

$$b_r \left( 2 \left\lceil \log_{M-1}(b_r/M) \right\rceil + 1 \right)$$
Join Operation

Several different algorithms to implement joins
- Block nested-loop join
- Indexed nested-loop join
- Merge-join
- Hash-join
Block Nested-Loop Join

\[
\text{for each block } B_r \text{ of } r \text{ do begin} \\
\text{for each block } B_s \text{ of } s \text{ do begin} \\
\text{for each tuple } t_r \text{ in } B_r \text{ do begin} \\
\text{for each tuple } t_s \text{ in } B_s \text{ do begin} \\
\text{Check if } (t_r, t_s) \text{ satisfy the join condition} \\
\text{if they do, add } t_r \cdot t_s \text{ to the result.} \\
\text{end} \\
\text{end} \\
\text{end} \\
\text{end} \\
\text{end}
\]

- \( r \): outer relation
- \( s \): inner relation
- \( \text{cost} = b_r \ast b_s + b_r, \)
  - Assuming one buffer for each
Block Nested-Loop Join

- Requires no indices
- Can be used with any join condition
- Cost = \( b_r \times b_s + b_r \)
  - Which relation shall be out relations?
- How to allocate \( M \) buffers to inner loop/out loop/result?
Indexed Nested-Loop Join

- Index available on the inner relation’s join attribute
  - Cost = $b_r + n_r \times c$
    - Where $c$ is the cost of traversing index and fetching all matching $s$ tuples for one tuple or $r$

- Index available on the outer relation’s join attribute
  - We do not use it

- If indices are available on join attributes of both $r$ and $s$
  - Use the relation with fewer tuples as the outer relation.
Exercise

- Compute \textit{depositor} $\bowtie$ \textit{customer}, with \textit{depositor} as the outer relation.
- \textit{Customer} has a secondary B$^+$-tree index on \textit{customer-name}
  - Blocking factor 20 keys
- $\#\textit{customer} = 400b/10,000t$  $\#\textit{depositor} =100b/5,000t$
- Block nested loops
  - $400\times100 + 100 = 40,100$
- Indexed nested loops
  - $100 + 5000 \times 5 = 25,100$ disk accesses.
  - CPU cost likely to be less than that for block nested loops join
Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).

2. Merge the sorted relations to join them
   1. Join step is similar to the merge stage of the sort-merge algorithm.
   2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched

\[
\text{Cost} = br + bs + \text{the cost of sorting if relations are unsorted}
\]
Merge-Join

- Can be used only for equi-joins and natural joins
- Why not on inequality, great than, less than joins?
Hybrid Merge-join

- If one relation is sorted, and the other has a secondary B⁺-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B⁺-tree.
  - Sort the result on the addresses of the unsorted relation’s tuples.
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples.
  - Sequential scan more efficient than random lookup.
Exercise

- Compute \( \text{deposito} r \times \text{customer} \), with \( \text{deposito} r \) as the outer relation.

- \( \text{Customer} \) has a secondary B\(^+\)-tree index on \( \text{customer-name} \)
  - Blocking factor 20 keys

- \#\( \text{customer} \) = 400b/10,000t  \#\( \text{deposito} r \) = 100b/5,000t

- Merge join
  - \( \log(400) + \log(100) + 400 + 100 = 516 \)

- Hybrid merge join
  - \( \log(100) + 100 + 500 + \log(600) + \text{number of blocks containing matching depositor records} \)