Query Processing

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

User

```
select * from R, S where ...
```

Query Parser

Resolve the references, Syntax errors etc. Converts the query to an internal format

relational algebra like

Query Optimizer

Find the best way to evaluate the query

Which index to use? What join method to use?

Query Processor

Read the data from the files

Do the query processing

joins, selections, aggregates

...
Selection Operation

- SELECT * FROM person WHERE SSN = “123”

Option 1: Sequential Scan
- Read the relation start to end and look for “123”
  - Does not have to be sorted (not true for the other options)
- Cost?
  - Let $b_r = \text{Number of relation blocks}$
  - Then:
    - 1 seek and $b_r$ block transfers
  - So:
    - $t_s + b_r \cdot t_r \text{ sec}$
- Improvements:
  - If SSN is a key, then can stop when found
    - So on average, $b_r/2$ blocks accessed

Option 2: Binary Search:
- Pre-condition:
  - The relation is sorted on SSN
  - Selection condition is an equality
    - E.g. can’t apply to “Name like ‘%424%’”
- Do binary search
  - Cost of finding the first tuple that matches
    - $\lceil \log_2(b_r) \rceil \cdot (t_r + t_s)$
    - All I/Os are random, so need a seek for all
      - The last few are short hops, but all seeks counted the same
  - Not quite: What if 10000 tuples match the condition? (SSN not a key)
    - Incurs additional cost
Selection Operation

- SELECT * FROM person WHERE SSN = “123”

- Option 3: **Use Index**
  - Pre-condition:
    - *An appropriate index must exist*
  - Use the index
    - Find the first leaf page that contains the search key
    - Retrieve all the tuples that match by following the pointers
      - If primary index, the relation is sorted by the search key
        - Go to the relation and read blocks sequentially
      - If secondary index, must follow all pointers using the index

---

**Selection w/ B+-Tree Indexes**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Cost of Finding the First Leaf</th>
<th>Cost of Retrieving the Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary index, search on a key, equality</td>
<td>$h_i \cdot (t_T + t_S)$</td>
<td>$1 \cdot (t_T + t_S)$</td>
</tr>
<tr>
<td>primary index, search on non-key, equality</td>
<td>$h_i \cdot (t_T + t_S)$</td>
<td>$1 \cdot (t_T + t_S) + (b - 1) \cdot t_T$</td>
</tr>
<tr>
<td>secondary index, key, equality</td>
<td>$h_i \cdot (t_T + t_S)$</td>
<td>$1 \cdot (t_T + t_S)$</td>
</tr>
<tr>
<td>secondary index, not a key, equality</td>
<td>$h_i \cdot (t_T + t_S)$</td>
<td>$n \cdot (t_T + t_S)$</td>
</tr>
</tbody>
</table>

$h_i = \text{height of the index}$

*Note:* primary == sorted

$b = \text{number of pages that contain the matches}$

This can be bad
Selection Operation

- Selections involving ranges
  - `select * from accounts where balance > 100000`
  - `select * from matches where matchdate between '10/20/06' and '10/30/06'`
  - Option 1: Sequential scan
  - Option 2: Using an appropriate index
    - Can’t use hash indexes for this purpose

- Complex selections
  - **Conjunctive**: `select * from accounts where balance > 100000 and SSN = "123"`
  - **Disjunctive**: `select * from accounts where balance > 100000 or SSN = "123"`
  - Option 1: Sequential scan
  - **Option 2 (Conjunctive only)**: Using an appropriate index on one of the conditions
    - E.g. Use SSN index to evaluate SSN = "123". Apply the second condition to the tuples that match
    - Or do the other way around (if index on balance exists)
    - Which is better?
  - **Option 3 (Conjunctive only)**: Choose a multi-key index
    - Not commonly available
Selection Operation

- Complex selections
  - **Conjunctive**: `select * from accounts where balance > 100000 and SSN = "123"`
  - **Disjunctive**: `select * from accounts where balance > 100000 or SSN = "123"

- **Option 4**: Conjunction or disjunction of record identifiers
  - Use indexes to find all RIDs that match each of the conditions
  - Do an **intersection** (for conjunction) or a **union** (for disjunction)
  - Sort the records and fetch them in one shot
  - Called “Index-ANDing” or “Index-ORing”

- Heavily used in commercial systems

Example

"select * from R where a = 10 and b = 20."

- R occupies 1 million blocks on disk
- secondary indexes of height 4 on both R.a and R.b.
  - Num tuples in R with R.a = 10 is 2000,
  - with R.b = 20 is 4000,
  - and both R.a = 10 & R.b = 20 is 100.

How many blocks transferred for each of the following options? For all the indexes, assume:

- 500 pointers (to the actual records) each leaf
- 100 records per block

(1) Use index on R.a to fetch tuples matching R.a = 10, and check the condition in memory,

- 2000 blocks for the tuples
- 4 blocks for the index (including first leaf)
- 3 more blocks additional leaves
- 2007
Example

"select * from R where a = 10 and b = 20."

- R occupies 1 million blocks on disk
- secondary indexes of height 4 on both R.a and R.b.
  - Num tuples in R with R.a = 10 is 2000,
  - with R.b = 20 is 4000,
  - and both R.a = 10 & R.b = 20 is 100.

How many blocks transferred for each of the following options? For all the indexes, assume:
- 500 pointers (to the actual records) each leaf
- 100 records per block

(2) Use index on R.b to fetch tuples matching R.b = 20, and check the other condition in memory,
- 4000 blocks for the tuples
- 4 blocks for the index (including first leaf)
- 7 more blocks additional leaves
- 4011

(3) Do "Index-ANDing" using both indexes.
- 4 blocks for first index
- 3 more for rest of leaves
- 7 more for rest of leaves
- 100 for data (each tuple probably on different block)

118
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?
    - \( n_r + b_r \)

---

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 -- \( R \): \( n_r = 10,000 \), \( S \): \( n_s = 5000 \)
  - Number of blocks -- \( R \): \( b_r = 400 \), \( S \): \( b_s = 100 \)
- **Then for \( R \) “outer relation”:**
  - blocks transferred: \( n_r \times b_s + b_r = 10000 \times 100 + 400 = 1,000,400 \)
  - seeks: 10400
  - time: \( 1000400 \times t_r + 10400 t_s = 1000400(1\text{ms}) + 10400(4\text{ms}) = 141.6 \text{ sec} \)
- **What if \( S \) “outer relation”?
  - \( 5000 \times 400 + 100 = 2,000,100 \) block transfers,
  - 5100 seeks
  - \( = 2000100 \times t_r + 5100 t_s = 220.4 \text{ sec} \)

*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
  - cost: 58 msecs

*Orders of magnitude difference*