Query Processing

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

User

**Query Parser**

**Query Optimizer**

**Query Processor**

R, B+Tree on R.a

S, Hash Index on S.a

---

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

*relational algebra like*

Find the *best* way to evaluate the query
Which index to use ?
What join method to use ?

---

Read the data from the files
Do the query processing

*joins, selections, aggregates*

---

“Cost”

- Complicated to compute
- We will focus on disk:
  - Number of I/Os ?
    - Not sufficient
    - Number of seeks matters a lot… why ?
  - \( t_T \) – time to transfer one block
  - \( t_S \) – time for one seek
  - Cost for \( b \) block transfers plus \( S \) seeks
    \[ b * t_T + S * t_S \]
  - Measured in *seconds*
Selection Operation

- SELECT * FROM person WHERE SSN = “123”
- **Option 1: Sequential Scan**
  - Read the relation start to end and look for “123”
  - Can always be used (not true for the other options)
  - Cost?
    - Let \( b_r \) = Number of relation blocks
    - Then:
      - 1 seek and \( b_r \) block transfers
    - So:
      - \( t_s + b_r \cdot t_T \) sec
    - Improvements:
      - If SSN is a key, then can stop when found
      - So on average, \( b_r/2 \) blocks accessed

Selection Operation

- SELECT * FROM person WHERE SSN = “123”
- **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
      - \( \lceil \log_2(b_r) \rceil \cdot (t_T + t_s) \)
      - All I/Os are random, so need a seek for all
      - The last few are short hops, but we ignore such small effects
  - Not quite: What if 10000 tuples match the condition?
    - 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></th>
<th>cost of finding the first leaf</th>
<th>cost of retrieving the tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary index, candidate key, equality</td>
<td>$h_i \times (t_T + t_S)$</td>
<td>$1 \times (t_T + t_S)$</td>
</tr>
<tr>
<td>primary index, not a key, equality</td>
<td>$h_i \times (t_T + t_S)$</td>
<td>$1 \times (t_T + t_S) + (b - 1) \times t_T$</td>
</tr>
<tr>
<td>Note: primary == sorted</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$b = number of pages that contain the matches$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>secondary index, candidate key, equality</td>
<td>$h_i \times (t_T + t_S)$</td>
<td>$1 \times (t_T + t_S)$</td>
</tr>
<tr>
<td>secondary index, not a key, equality</td>
<td>$h_i \times (t_T + t_S)$</td>
<td>$n \times (t_T + t_S)$</td>
</tr>
<tr>
<td>$n = number of records that match$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>This can be bad</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\text{why?}$

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

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 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

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 ($N \geq M$)**

- **Initial relation**
  - Creates runs

- **Runs**
  - Merge pass 1

- **Sorted output**
  - Merge pass 2

$M = 3$

$N = 12$
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 don’t count write cost
      - output may be pipelined (sent via memory to parent operation)

Thus total number of disk transfers for external sorting:

$$b_r \cdot (2 [\log_{M-1}(b_r/M)] + 1)$$

**Seeks:**

$$2 [b_r/M] + [b_r/b_b] (2 [\log_{M-1}(b_r/M)] - 1)$$

$b_b$ is #blocks read at a time, and how many output blocks needed.

---

Example: External Sorting Using Sort-Merge ($N \geq M$)

- **$M = 3$**
- **$N = 12$**

- $b_r \cdot (2 [\log_{M-1}(b_r/M)] + 1)$ blocks
- seeks:

$$2 [b_r/M] + [b_r/b_b] (2 [\log_{M-1}(b_r/M)] - 1)$$

 initial relation | runs | create runs | merge pass–1 | merge pass–2 | sorted output
---|---|---|---|---|---
 g 24 | d 31 | a 19 | a 14 | a 19 |
 a 24 | g 24 | b 14 | c 33 | b 14 |
 d 31 | b 14 | c 33 | d 31 | c 33 |
 c 33 | e 16 | e 16 | d 7 | d 7 |
 b 14 | e 16 | g 24 | d 21 | m 3 |
 e 16 | | | d 7 | e 16 |
 r 16 | d 7 | a 14 | m 3 | g 24 |
 d 21 | m 3 | p 2 | p 2 | p 2 |
 m 3 | r 16 | | | |
External Merge Sort (Cont.)

Example:
- For \( b_r = 12, M = 3 \)
- Disk transfers = \( 12(2 \lceil \log_2(12/3) \rceil + 1) = 60 \)
- Seeks = \( 2 \lceil 12/3 \rceil + 12 ((2 \lceil \log_2(12/3) \rceil - 1) = 8 + 36 = 44 \)

Query Processing

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

- \( \text{select } * \text{ from } R, S \text{ where } R.a = S.a \)
  - Called an “equi-join”
- \( \text{select } * \text{ from } R, S \text{ where } |R.a - S.a| < 0.5 \)
  - 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 \( S \): \( n_r \ast b_s \)
  - Seeks ?
    - \( n_r + b_r \)
Nested-loops Join

- **Case 1:** Minimum memory required = 3 blocks
  - Blocks transferred: \( n_r \cdot b_s + b_r \)
  - 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: \( n_r \cdot b_s + b_r = 10000 \cdot 100 + 400 = 1,000,400 \)
  - seeks: 10400
  - time: \( 1000400 \cdot t_r + 10400 \cdot t_s = 1000400(0.1\text{ms}) + 10400(4\text{ms}) = 1020.8 \text{ sec} \)
- **What if \( S \) outer relation?**
  - \( 5000 \cdot 400 + 100 = 2,000,100 \) block transfers,
  - 5100 seeks
  - \( = 2000100 \cdot t_r + 5100 \cdot t_s = 2041.7 \text{ sec} \)

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

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

Block Nested-loops Join

- **Case 1: Minimum memory required = 3 blocks**
  - Blocks transferred: $b_r * b_s + b_r$
  - Seeks: $2 * b_r$

- **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...
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: $2 * b_r / 48$

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 (t_T + t_S) + n_r * c`
  - `c == the cost of index access`
    - Computed using the formulas discussed earlier

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 \( 100 + 5000 * (4 + 1) = 25,100 \), each w/ seek and transfer
- Assume B*-tree is on `S`: height = 3
  - Cost is \( 400 + 10000 * (3+1) = 40,400 \), each w/ seek and transfer
Index Nested-loops Join

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

- **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`