Administriva

- Lab 2
  - Part 2 due tomorrow
  - Final version due next Wednesday
- Problem sets
  - PS 5 due Thursday
  - PS 6 out this Thursday
- Midterm
  - Next class: review session
  - Midterm out after next class, due 2/28 (no class that day)

Goals for Today

- Discuss algorithms for implementing query plan operators: selection, joins
  - Reason about cost in I/Os
- Understand how external sorting and hashing can be used for these algorithms
- Prepare for final exercises of Lab 2

Logical Plan to Physical Plan

- Logical query plan partially shows us how to evaluate query
  - Missing: choice of specific algorithm for executing operators

Access path choice: Sequential scan? Indexes?

Join algorithm choice:
- Simple nested loop
- Index nested loop
- Sort-merge?
- ...

Selections: how to apply multiple constraints?
Projections: duplicate elim.?
Review: Relational Operations

- Finding algorithms for:
  - Selection (\( \sigma \))
  - Projection (\( \pi \))
  - Cross-product and Join (\( \times \))
  - Set-difference (\( - \))
  - Union (\( \cup \))
  - Aggregation (SUM, MIN, etc.) and GROUP BY

- After we cover choices for algorithms, we will discuss how to optimize queries formed by composing operators

Schema for Examples

Sailors (\( sid \): integer, \( sname \): string, \( rating \): integer, \( age \): real)
Reserves (\( sid \): integer, \( bid \): integer, \( day \): date, \( rname \): string)

- Similar to old schema; \( rname \) added for variations.

- Reserves:
  - Each record is 40 bytes long
  - 100 record per page
  - \( M = 1000 \) pages

- Sailors:
  - Each record is 50 bytes long,
  - 80 record per page
  - \( N = 500 \) pages

Simple Selections

- Of the form \( \sigma_{R.\text{attr} \, \text{op} \, \text{value}}(R) \)

  - Size of result approximated as \( \text{size of } R \times \text{reduction factor} \)
    - “Reduction factor” also called selectivity
    - Statistics in Catalog can help us estimate

  - How best to execute a selection? Depends on:
    - What indexes/access paths are available
    - Expected size of the result
      (in terms of number of tuples and/or number of pages)

General Themes

- What can we do when all data cannot fit in RAM
  - Out-of-core algorithms
    - Conceptually: form partitions to get “like” things together

- Factors influencing operator cost:
  - Input size (number of pages)
  - Indexes available
  - Buffer pool space

Review: Relational Operations

Last time: sorting or hashing to remove duplicates when projecting

SELECT * FROM Reserves R WHERE R.bid < 100;
Simple Selections: not using indexes (Review)

- **With no index, unsorted:**
  - Must scan the whole relation
  - For Reserves = cost is M (#pages in R) \(\rightarrow\) 1000 I/Os.

- **With no index, sorted:**
  - Cost of binary search + number of pages containing results
  - For Reserves = \(\log_2 1000\) I/Os + \([\text{selectivity} \times \text{#pages}]\)

Using an Index for Selections

- Cost depends on # qualifying tuples and if index clustered
  - Cost has two components:
    - Finding qualifying data entries (typically small, e.g., finding leaf in tree)
    - Plus cost of retrieving records (could be large without clustering)

- For example query on “Reserves”, if 10% of tuples qualify:
  (result size estimate = 100 pages or 10,000 tuples)

Selections using Index (cont)

- **Important refinement for unclustered indexes:**
  1. Find qualifying data entries.
  2. Sort the record ids of the data records to be retrieved.
  3. Fetch pages in order!
     - Ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).

When using this refinement, is using the unclustered index ever better than sequential scan?

General Selection Conditions

```
SELECT *
FROM Reserves R
WHERE R.bid = 103 AND R.sid = 42;
```

- **A B+-tree index matches** (a conjunction of) terms if the term(s) involve only attributes in a *prefix* of the search key.
  - E.g., Index on \(<a, b, c>\) matches predicate “\(a=5 \text{ AND } b=3\)”
  - but not “\(b=3\)”

- For **Hash index:** index must involve all attributes in search key

Why?
General Selections: Two Approaches

• What if several indexes exist that could be used?

• Approach 1: pick one index to use
  – Find the most selective access path, retrieve tuples using it, then apply the other conditions

  Most selective access path: Index with file scan estimated to require fewest page I/Os

• Approach 2: use multiple indexes
  – To use two or more matching indexes (Alt 2 or 3 for data entries):
    – Get sets of record ids of data records using each matching index.
    – Then intersect these sets of rids.
    – Retrieve the records and apply any remaining conditions

  Example: day < 8/9/94 AND bid=103 AND sid=42
  Suppose have B+ tree index on day and an index on sid
  – Intersect: rids using index on day with rids using index on sid
  – Then check bid=103

Exercise: Selection

• Exercise 2
  I. B+tree on <rname, day>
  II. B+tree on <day, rname>
  III. Hash index on <day, rname>

• Disjunction:
  – if all conditions have an index, use the union of rids!
  – But if even one of them does not have index, have to do sequential scan anyway

Join Operators

• Joins are a very common query operation!

• Joins can be very expensive:
  – Consider an inner join of R and S each with 1M records
    How many tuples in the answer (worst case)?

• Two main classes of JOIN algorithms:
  – Algorithms that effectively enumerate cross product
  – Algorithms that avoid cross product by getting “like” partitions together
Equality Joins on One Join Column

\[
\text{SELECT * } \\
\text{FROM Reserves R, Sailors S } \\
\text{WHERE R.sid=S.sid}
\]

• Relation info:
  – M = 1000 pages in R, \( t_R = 100 \) tuples per page.
  – N = 500 pages in S, \( t_S = 80 \) tuples per page.
  – In examples, R is Reserves and S is Sailors.

• "Cost metric": # of I/Os
  (We will ignore cost of final output from query)

Simple Nested Loops Join

\[
\text{foreach tuple } r \text{ in } R \text{ do} \\
\quad \text{foreach tuple } s \text{ in } S \text{ do} \\
\qquad \text{if } r_i = s_j \text{ then add } <r, s> \text{ to result}
\]

• For each tuple in the outer relation R, we scan the entire inner relation S.
  – Cost: \( (t_R \times M) \times N + M \)
    \( = 100,000 \times 500 + 1000 \) I/Os.

• What if smaller relation (S) was outer?
  – \( (t_S \times N) \times M + N \)
    \( = 40,000 \times 1000 + 500 \) I/Os.

Page-Oriented Nested Loops Join

\[
\text{foreach page } p_R \text{ in } R \text{ do} \\
\quad \text{foreach page } p_S \text{ in } S \text{ do} \\
\quad \quad \text{foreach tuple } r \text{ in } p_R \text{ do} \\
\quad \quad \quad \text{foreach tuple } s \text{ in } p_S \text{ do} \\
\qquad \text{if } r_i = s_j \text{ then add } <r, s> \text{ to result}
\]

• For each page of R, get each page of S, and write out matching pairs of tuples \(<r, s>\), where \( r \) is in R-page and S is in S-page.

• What is the cost of this approach? (Try Exercise 3)

• With R as outer, cost = \( M \times N + M = 1000 \times 500 + 1000 \)
  – If smaller relation (S) is outer, cost = \( 500 \times 1000 + 500 \)

Block Nested Loops Join

• Page-oriented NL doesn’t use all available buffer frames!

• Alternative approach:
  – Use one page as an input buffer for scanning the inner S,
  – one page as the output buffer
  – and use all remaining pages to hold block of outer R

• For each block of R, scan through each page of S for matches
Block Nested Loop Join: Examples

- **Cost:** Scan of outer + # outer blocks * scan of inner
  
  \# outer blocks = ceiling(\# pages of outer/blocksize)

- With Reserves (R) as outer, and 100 pages/block:
  - Scanning R is 1000 I/Os; a total of 10 blocks.
  - Per block of R, scan Sailors (S); 10*500 I/Os.

- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, scan Reserves: 5*1000 I/Os.

Avoiding Cross-product

- Simple, Page-oriented, and Block Nested-loop join algorithms **effectively enumerate the cross-product**
  
  - every pair of tuples is compared

- Next: algorithms that avoid cross-product (for equality joins)
  
  - tuples in the two relations can be thought of as belonging to partitions

Exercise 4: Index Nested Loops

- Have Hash-index (Alt. 2) on sid of Sailors (as inner)
  Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  
  - For each Reserves tuple:
    - 1.2 I/Os to get data entry in index,
    - plus 1 I/O to get [the exactly one] matching Sailors tuple
  
  Total cost: 1000 + 2.2 * 100,000 = 221,000 I/Os

- Hash-index (Alt. 2) on sid of Reserves (as inner):
  
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  
  - For each Sailors tuple:
    - 1.2 I/Os to find index page with data entries,
      - plus cost of retrieving matching Reserves tuples
    
    Assuming uniform distribution, 2.5 reservations per sailor
    (100,000 tuples in R / 40,000 tuples in S).
  
  - Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered
Sort-Merge Join \((R \bowtie S)\)

- Sort \(R\) and \(S\) on the join column, then scan them to do a “merge” (on join field), and output result tuples.

- Particularly useful if
  - one or both inputs are already sorted on join field(s)
  - output is required to be sorted on join field(s)

Example of Sort-Merge Join

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>

Example of Sailors (outer)

Example of Reserves (inner)

Reflection: Sort-Merge Join

- Outer relation is scanned once

- Each partition of Inner relation is scanned once per matching Outer tuple
  - “Merge” phase can require some back tracking if duplicate values appear in join column
  - Multiple scans of an Inner partition will probably find needed pages in buffer pool

Hash-Join

(this variant: “Grace Hash Join”)

- Partition both relations on the join attributes using hash function \(h\)
- \(R\) tuples in partition \(R_i\) will only match \(S\) tuples in partition \(S_i\).
- For each partition \(i\)
  - Read in all of \(R_i\)
  - Hash \(R_i\) on \(h_2\)
  - Scan through pages of \(S_i\) probing hash table for matches

Why?
Aggregation Operations
(AVG, MIN, etc.)

• Without grouping:
  – In general, requires scanning the relation.
  – Given a tree index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan.

• With grouping:
  – Sort on group-by attributes, then scan relation and compute aggregate for each group.
  – Similar approach based on hashing on group-by attributes.
  – Can use a tree index whose search key “covers” all attributes in SELECT, WHERE and GROUP BY clauses.

SimpleDb: Insert and Delete

• Insert operator
  – Inserts tuples it gets from its child into a given relation
  – Output: number of tuples inserted
    ```sql
    INSERT INTO Students(sid, name, gpa)
    VALUES (42, “Alice”, 3.5);
    ```

• Delete operator
  – Deletes the tuples it gets from its child
  – Output: number of tuples deleted
    ```sql
    DELETE FROM Students
    WHERE gpa < 2.0;
    ```

SimpleDb: Buffer Pool Eviction

• Implement a page eviction policy
  – Update getPage() to evict if necessary

• Flushing pages
  – Before a dirty page is evicted, its changes must be written to disk!
  – However, calling flushPage() should not evict pages!!