Class Agenda

• Last time:
  – Query evaluation techniques; external sorting

• Today:
  – Finish with external sorting
  – Physical query operators

• Reading
  – Chapters 13 and 14 of Ramakrishnan and Gehrke (or Chapter 13 of Silberschatz et al)
Announcements

Grades for Part 1: Monday

**Quiz #1 in class next Wednesday** (now reflected on web page); review session in class Monday

**Quiz #2** (along with "Awards Ceremony") **will be during final exam slot**
External Sorting, continued
Using B+ Trees for Sorting

- Scenario: table to be sorted has B+ tree index on sorting column(s)
- Idea: can retrieve records in order by traversing leaf pages
- Is this a good idea?
- Cases to consider:
  - B+ tree is clustered \textit{Good idea!}
  - B+ tree is not clustered \textit{Could be a very bad idea!}
Clustered B+ Tree Used for Sorting

- Cost: root to the leftmost leaf, then retrieve all leaf pages (index is clustered)
- Each page fetched just once
- Always better than external sorting!
Unclustered B+ Tree Used for Sorting

- Leaves of tree have record ids, rather than records themselves
- In worst case, one I/O per data record!
### External Sorting vs Unclustered Index

<table>
<thead>
<tr>
<th># of data pages</th>
<th>Unclustered index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000</td>
</tr>
</tbody>
</table>

- $p = \#$ of records per page
- $B = 1000$ and block size = 32 for external sorting
- $p = 100$ is the more realistic value
Summary of External Sorting

• External sorting is important; DBMS may dedicate part of buffer pool for sorting!

• External merge sort minimizes disk I/O cost
  
  – Pass 0: produces sorted runs of size $B$ (# of buffer pages)
  
  – # of runs merged at a time depends on $B$ and block size
  
  – Larger block size means less I/O cost per page
  
  – Larger block size means smaller # runs merged
  
  – In practice, # of runs rarely more than 2 or 3
• 2-way merge sort can be generalized to $n$-way merge sort, using as many internal buffer pages as we have available.
Physical Relational Operators, Part 1: Joins
Relational Operations

• We will consider how to implement:
  – **Selection** ($\sigma$) Selects a subset of rows from relation
  – **Projection** ($\pi$) Deletes/reorders columns from relation
  – **Join** ($\&$) Allows us to combine two relations
  – **Difference** (−) Tuples in one relation, but not the other
  – **Union** (U) Tuples in either relation
  – **Aggregation** SUM, MIN, etc. and GROUP BY

• Since each operation returns a relation, operations can be composed.

• After we cover the operations in isolation, we will discuss how to optimize queries formed by composing them
Schema for Running Examples

Sailors($sid$: integer, $sname$: string, $rating$: integer, $age$: float)

Reserves($sid$: integer, $bid$: integer, $day$: date, $rname$: string)

- Reserves: each tuple is 40 bytes long, 100 tuples per page, 1000 pages
- Sailors: each tuple is 50 bytes long, 80 tuples per page, 500 pages
Equality Joins With One Join Column

select *
from Reserves R, Sailors S
where R.sid = S.sid

• Common! Must be carefully optimized. \( R \times S \) is large, so \( R \times S \) followed by selection is inefficient

• Assume: \( M \) tuples in \( R \), \( p_R \) tuples per page, \( N \) tuples in \( S \), \( p_S \) tuples per page
  
  — In our examples, \( R \) is Reserves and \( S \) is Sailors

• Will consider more complex join conditions later

• **Cost metric:** # of I/Os
Simple Nested Loops Join

for each tuple $r$ in $R$ do
  for each tuple $s$ in $S$ do
    if $r$ and $s$ agree on join attribute then
      add $<r,s>$ to result

• For each tuple in the outer relation $R$, we can the entire inner relation $S$
  
  – Cost: $M + p_R \times M \times N = 1000 + 100\times1000\times500$ I/Os

• Page-oriented nested loops join: 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
  
  – Cost: $M + M \times N = 1000 + 1000\times500$
Index Nested Loops Join

for each tuple \( r \) in \( R \) do
  for each tuple \( s \) in \( S \) do
    if \( r \) and \( s \) agree on join attribute then
      add \( <r,s> \) to result

• If there is an index on the join attribute of one relation (say \( S \)), can make it the inner and exploit the index
  – Cost: \( M + ((M \times p_R) \times \text{cost of finding matching } S \text{ tuples}) \)

• For each \( R \) tuple, cost of probing \( S \) index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding \( S \) tuples depends on clustering
  – Clustered index: usually 1 I/O per group of tuples with a given key;
    unclustered: up to 1 I/O per tuple in group of tuples with a given key
Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and all remaining pages to hold block of outer R
  - For each matching tuple r in R-block, s in S-page, add <r,s> to result. Then read next R-block, scan S, and repeat.
Sort-Merge Join

• Sort $R$ and $S$ on the join attribute, then scan them to do a \textit{merge} (on join attribute), and output result tuples
  
  – Advance scan of $R$ until current $R$-tuple $\geq$ current $S$-tuple, then advanced scan of $S$ until current $S$-tuple $\geq$ current $R$-tuple; do this until current $R$-tuple = current $S$-tuple
  
  – At this point, $R$-tuple \textit{matches} current $S$-tuple (and all following $S$-tuples with same value); output $<r,s>$ for all pairs of such tuples
  
  – Then resume scanning $R$ and $S$

• $R$ is scanned once; each $S$ "group" is scanned once per matching $R$ tuple.
Example of Sort-Merge Join

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>puppy</td>
<td>9</td>
<td>35.0</td>
<td>103</td>
<td>11/3/96</td>
<td>puppy</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>

- Cost: $M \log M + N \log N + \sim (M + N)$
  - In worst case $M + N$ could actually be $M*N$, but unlikely
Refinement of Sort-Merge Join

We can combine the merging phases in the *sorting* of R and S with the merging required for the join.

- With $B > \sqrt{L}$, where $L$ is the size of the larger relation, using the sorting refinement that produces runs of length $2B$ in Pass 0, the number of runs is less than $B/2$.
- Allocate 1 page per run of each relation, and `merge’ while checking the join condition.
- Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
- In example, cost goes down from 7500 to 4500 I/Os.

In practice, cost of sort-merge join, like the cost of external sorting, is linear.
Hash Join

- Partition both relations using hash function $h$: $R$ tuples in partition $i$ will only match $S$ tuples in partition $i$.
- Read in a partition of $R$, hash it using $h'$ ($\neq h$!). Scan matching partition of $S$, search for matches.