Acknowledgements: portions based on slides by Raghu Ramakrishnan and Johannes Gehrke.
Class Agenda

• Last time:
  – Overview of DavisDB project, Part 2: indexing

• Today:
  – Query evaluation techniques: sorting

• Reading
  – Chapters 12 and 13 of Ramakrishnan and Gehrke (or Chapter 13 of Silberschatz et al)
Announcements

**Code review sign-up sheet** posted (see email I sent out for link); code reviews happening today through Monday

**Repository updates:** TestIX.cpp (sample tests for indexing); page file manager bugfixes; (not quite) final version of TestRM.cpp*

**Grades** for Part 1: Friday?

**Discussion section** Friday @11am: **B+ tree jam session**

Quiz #1 in class next Wednesday
Overview of Query Evaluation Techniques

Background material for Part 4 of the DavisDB project; some concepts we saw in Lecture 7 include:

• **evaluation plan** – relational algebra query drawn as a tree;

• **annotated evaluation plan** – each relational operator (e.g., "join") is annotated with the **physical operator** that will be used to perform the operation (e.g., "index nested loops join")

• **query optimizer** – takes a SQL query, produces an efficient annotated evaluation plan

• **query execution engine** – executes the annotated evaluation plan
### Logical versus Physical Operators

<table>
<thead>
<tr>
<th>Logical operator</th>
<th>Physical operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>join: ( E_1 \times E_2 )</td>
<td>nested loops join, index nested loops join, sort-merge join, ...</td>
</tr>
<tr>
<td>projection: ( \pi(E) )</td>
<td>projection</td>
</tr>
<tr>
<td>predicate: ( R )</td>
<td>file scan, index scan, ...</td>
</tr>
<tr>
<td>selection: ( \sigma(E) )</td>
<td>selection</td>
</tr>
<tr>
<td>selection w/base predicate: ( \sigma(R) )</td>
<td>file scan with condition, index scan with condition, ...</td>
</tr>
</tbody>
</table>

- File scan (with condition): RecordFileScan (DavisDB Part 1)
- Index scan (with condition): IndexScan (Part 2)
  - Also underlies index nested loops join
- Others will be implemented in QueryEngine (Part 4)
Recall: Sort-Merge Join of $R$ and $S$

- Sort $R$ and $S$ on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
  - Advance scan of $R$ until current $R$-tuple $\geq$ current $S$ tuple, then advance scan of $S$ until current $S$-tuple $\geq$ current $R$ tuple; do this until current $R$ tuple $=$ current $S$ tuple.
  - At this point, all $R$ tuples with same value in $R_i$ (current $R$ group) and all $S$ tuples with same value in $S_j$ (current $S$ group) match; 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. (Multiple scans of an $S$ group are likely to find needed pages in buffer.)
Example of Sort-Merge Join

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</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>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>

- Cost: $M \log M + N \log N + (M+N)$
  - The cost of scanning, $M+N$, could be $M*N$ (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.
Something to Consider in Part 2 (Indexing)

• In Part 4, **nested loops join** and **index nested loops join** will be the only join algorithms you will be required to implement.

• Sort-merge join will be optional (XC), *but*, here's something to do in Part 2 that will make it easier.

• Scan of B+ tree: **required** to return all record ids matching condition; **not required** to return them in order!

• May be a little extra work to have your scan return them in order, depending on details of your implementation.

• *But* this will let you use the index to do the sort for sort-merge join, if $R$ and $S$ are both indexed on the join attribute.
  – We'll look at this again in a few slides.
Plan for Upcoming Lectures

• Rest of today: we'll talk about **external sorting**, needed for sort-merge join, duplicate elimination, ...

• Next lecture: we'll focus on the other physical query operators

• Subsequent lectures: generating physical plans (annotated evaluation plans) from logical plans (evaluation plans, aka relational algebra)