Evaluation of Relational Operations

CMPSCI 645
Mar 9th and 11th, 2010
Relational Operations

- We will consider how to implement:
  - **Selection** ($\sigma$) Selects a subset of rows from relation.
  - **Projection** ($\pi$) Deletes unwanted columns from relation.
  - **Join** ($\bowtie$) Allows us to combine two relations.
  - **Set-difference** ($\setminus$) Tuples in reln. 1, but not in reln. 2.
  - **Union** ($\cup$) Tuples in reln. 1 and in reln. 2.
  - **Aggregation** (SUM, MIN, etc.) and GROUP BY
  - **Order By** Returns tuples in specified order.

After we cover the operations, we will discuss how to optimize queries formed by composing them.
Outline

- Sorting
- Evaluation of joins
- Evaluation of other operations
Why Sort?

- A classic problem in computer science!
- Important utility in DBMS:
  - Data requested in sorted order (e.g., ORDER BY)
    - e.g., find students in increasing gpa order
  - Sorting useful for eliminating duplicates (e.g., SELECT DISTINCT)
  - Sort-merge join algorithm involves sorting.
  - Sorting is first step in bulk loading B+ tree index.
- **Problem**: sort 1Gb of data with 1Mb of RAM.
2-Way Sort: Requires 3 Buffers

- **Pass 1**: Read a page, sort it, write it.
  - only one buffer page is used
- **Pass 2, 3, …, etc.**:
  - three buffer pages used.

![Diagram](image-url)
Two-Way External Merge Sort

- Each pass we read + write each page in file: $2N$.
- $N$ pages in the file => the number of passes $= \lceil \log_2 N \rceil + 1$
- So total cost is: $2N \left( \lceil \log_2 N \rceil + 1 \right)$

Idea: Divide and conquer: sort subfiles and merge
More than 3 buffer pages. How can we utilize them?

To sort a file with $N$ pages using $B$ buffer pages:
- Pass 0: use $B$ buffer pages. Produce $\lfloor N / B \rfloor$ sorted runs of $B$ pages each.
- Pass 2, ..., etc.: merge $B-1$ runs.
Cost of External Merge Sort

- Number of passes: \(1 + \lceil \log_{B-1} \left( \frac{N}{B} \right) \rceil\)
- Cost = \(2N \times \# \text{ of passes}\)
- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: \(\lceil \frac{108}{5} \rceil = 22\) sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: \(\lceil \frac{22}{4} \rceil = 6\) sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages
## Number of Passes of External Sort

<table>
<thead>
<tr>
<th>N</th>
<th>B=3</th>
<th>B=5</th>
<th>B=9</th>
<th>B=17</th>
<th>B=129</th>
<th>B=257</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10,000,000</td>
<td>23</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
Replacement Sort

- Produces sorted runs as long as possible.
- Pick tuple $r$ in the current set with the smallest value that is ≥ largest value in output, e.g. 8 in the example.
- Fill the space in current set by adding tuples from input.
- Write output buffer out if full, extending the current run.
- Current run terminates if every tuple in the current set is smaller than the largest tuple in output.
- When used in Pass 0 for sorting, can write out sorted runs of size $2B$ on average.
**Blocked I/O for External Merge Sort**

- … longer runs often means fewer passes!
- Actually, we don’t do I/O a page at a time
- In fact, read a block of pages sequentially!
- Suggests we should make each buffer (input/output) be a block of pages.
  - But this will reduce fan-out during merge passes!
  - In practice, most files still sorted in 2-3 passes.
Double Buffering

- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block`.
  - Potentially, more passes; in practice, most files *still* sorted in 2-3 passes.

Diagram:
- Disk
- Main memory buffers: $B$
- $k$-way merge

INPUT 1 → OUTPUT
INPUT 1' → OUTPUT'
INPUT 2 → OUTPUT
INPUT 2' → OUTPUT'
INPUT $k$ → OUTPUT
INPUT $k'$ → OUTPUT'

b
block size

B main memory buffers, k-way merge
Sorting Records!

- Sorting has become highly competitive!
  - Parallel sorting is the name of the game ...

- Datamation sort benchmark: Sort 1M records of size 100 bytes
  - in 1985: 15 minutes
  - World records: 1.18 seconds (1998 record)
    - 16 off-the-shelf PC, each with 2 Pentium processor, two hard disks, running NT4.0.

- New benchmarks proposed:
  - Minute Sort: How many can you sort in 1 minute?
  - Dollar Sort: How many can you sort for $1.00?
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 left-most leaf, then retrieve all leaf pages (Alternative 1)

- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.

(always better than external sorting!)

Related Data Records
Unclustered B+ Tree Used for Sorting

- Alternative (2) for data entries; each data entry contains \( rid \) of a data record. In general, one I/O per data record!

Worse case I/O: \( pN \)

\( p \): # records per page

\( N \): # pages in file
## External Sorting vs. Unclustered Index

<table>
<thead>
<tr>
<th>N</th>
<th>Sorting</th>
<th>p=1</th>
<th>p=10</th>
<th>p=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>200</td>
<td>100</td>
<td>1,000</td>
<td>10,000</td>
</tr>
<tr>
<td>1,000</td>
<td>2,000</td>
<td>1,000</td>
<td>10,000</td>
<td>100,000</td>
</tr>
<tr>
<td>10,000</td>
<td>40,000</td>
<td>10,000</td>
<td>100,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>100,000</td>
<td>600,000</td>
<td>100,000</td>
<td>1,000,000</td>
<td>10,000,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td>8,000,000</td>
<td>1,000,000</td>
<td>10,000,000</td>
<td>100,000,000</td>
</tr>
<tr>
<td>10,000,000</td>
<td>80,000,000</td>
<td>10,000,000</td>
<td>100,000,000</td>
<td>1,000,000,000</td>
</tr>
</tbody>
</table>

* p: # of records per page
* B=1,000 and block size=32 for sorting
* p=100 is the more realistic value.
Summary

- 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 \( B \) (# buffer pages). Later passes: merge runs.
  - # 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.
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.
Outline

- Sorting
- Evaluation of joins
- Evaluation of other operations
Some Common Techniques

- Algorithms for evaluating relational operators use some simple ideas extensively:
  - **Indexing:** Can use WHERE conditions to retrieve small set of tuples (selections, joins)
  - **Iteration:** Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - **Partitioning:** By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

*Watch for these techniques as we discuss query evaluation!*
Schema for Examples

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

- **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

In algebra: $R \bowtie S$. Common relational operation!
- $R \times S$ is large; $R \times S$ followed by a selection is inefficient.
- Must be carefully optimized.

Assume: $M$ pages in $R$, $p_R$ tuples per page, $N$ pages in $S$, $p_S$ tuples per page.
- In our examples, $R$ is Reserves and $S$ is Sailors.

We will consider more complex join conditions later.

Cost metric: # of I/Os. We will ignore output costs.
Simple Nested Loops Join

foreach tuple r in R do
    foreach tuple s in S do
        if $r_i == s_j$ then add <r, s> to result

- For each tuple in the outer relation R, we scan the entire inner relation S.
  - Cost: $M + p_R \times M \times N = 1000 + 100 \times 1000 \times 500 = 1,000 + (5 \times 10^7)$ I/Os.
  - Assuming each I/O takes 10 ms, the join will take about 140 hours!
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: \( M + M \times N = 1000 + 1000 \times 500 = 501,000 \) I/Os.
  - Assuming each I/O takes 10 ms, the join will take about 1.4 hours.

- Choice of the smaller relation as the outer
  - If smaller relation (S) is outer, cost = \( 500 + 500 \times 1000 = 500,500 \) I/Os.
**Block Nested Loops Join**

- Take the **smaller** relation, say R, as **outer**, the other as inner.
- Use one buffer for scanning the inner S, one buffer for output, and use all remaining buffers 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 page in S, until S is finished.
  - Then read next R-block, scan S…
Examples of Block Nested Loops

- **Cost:** Scan of outer + #outer blocks * scan of inner
  - #outer blocks = ⌈# pages of outer / block size⌉
  - Given available buffer size B, block size is at most B-2.
  - M + N * ⌈M / B-2⌉

- With Sailors (S) as outer, let block be 100 pages of S:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
  - Total = 500 + 5 * 1000 = 5,500 I/Os.
  - (a little over 1 minute)
Index Nested Loops Join

foreach tuple r in R do
  foreach tuple s in S where r_i == s_j do
    add <r, s> to result

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: $M + ( (M * p_R) * \text{cost of finding matching S tuples})$

- 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 (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical).
  - Unclustered: up to 1 I/O per matching S tuple.
Examples of Index Nested Loops

- 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: 1000 + 100*1000*2.2 = 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. If uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os (cluster?).
  - Total: 500 + 80*500*(2.2~3.7) = 88,500~148,500 I/Os.
Sort-Merge Join \( (R \bowtie S) \) 

- (1) **Sort** \( R \) and \( S \) on the join column, (2) **Merge** them (on join col.), and output result tuples.

- **Merge**: repeat until either \( R \) or \( S \) is finished
  - **Scanning**: Advance scan of \( R \) until current \( R \)-tuple \( \geq \) current \( S \) tuple, advance scan of \( S \) until current \( S \)-tuple \( \geq \) current \( R \) tuple; do this until current \( R \) tuple = current \( S \) tuple.
  - **Matching**: Now 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.

- **\( R \) 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

- Cost: $M \log M + N \log N + (M+N)$
  - The cost of merging, $M+N$, could be $M*N$ (very unlikely!)
  - $M+N$ is guaranteed in foreign key join (why?)
  - As with sorting, $\log M$ and $\log N$ are small numbers, e.g., 3, 4.
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.
  
  $$(BNL\ cost: \ 2500\ (B=300),\ 5500\ (B=100),\ 15000\ (B=35))$$
**Hash-Join**

- **Partitioning**: Partition both relations using hash fn $h$: R tuples in partition $i$ will only match S tuples in partition $i$.

- **Probing**: Read in partition $i$ of R, build hash table on $R_i$ using $h2 (<> h!)$, scan partition $i$ of S, search for matches.
Observations on Hash-Join

- **# partitions ≤ B-1**, and size of largest partition ≤ B-2 to be held in memory. Assuming uniformly sized partitions, we get:
  - $M / (B-1) < (B-2)$, i.e., $B$ must be $> \sqrt{M}$
  - Hash-join works if the **smaller** relation satisfies above.

- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.

- If hash function $h$ does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.
Cost of Hash-Join

- Partitioning reads+writes both relns; 2(M+N). Probing reads both relns; M+N I/Os. The total is 3(M+N).
  - In our running example, a total of 4500 I/Os using hash join, less than 1 min (compared to 140 hours w. NLJ).

- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory both have a cost of 3(M+N) I/Os.
  - Hash Join superior on this count if relation sizes differ greatly. Assuming M<N, what if \( \sqrt{M} < B < \sqrt{N} \)? Also, Hash Join is shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.
General Join Conditions

- Equalities over several attributes (e.g., $R.sid = S.sid$ AND $R.rname = S.sname$):
  - For Index NL, build index on $<sid, sname>$ (if $S$ is inner); or use existing indexes on $sid$ or $sname$ and check the other join condition on the fly.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

- Inequality conditions (e.g., $R.rname < S.sname$):
  - For Index NL, need B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins (clustered index is much preferred).
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be a winner here.