Query Processing

Introduction to Databases
CompSci 316 Fall 2017

Announcements (Mon., Mar. 20)

- Homework #3
  - 3.1 and 3.2 are due on Wednesday March 22
- Project
  - Milestone 2 due next Monday March 27
  - Feedback posted on private piazza threads

Where are we?

- We are covering DB internals and query processing
- So far:
  - Index: mostly B+ tree
- Today:
  - finish query processing and join algorithms

Overview

- Many different ways of processing the same query
  - Scan? Sort? Hash? Use an index?
  - All have different performance characteristics and/or make different assumptions about data
- Best choice depends on the situation
  - Implement all alternatives
  - Let the query optimizer choose at run-time
  - Often not the “best choice”
  - Optimizer tries NOT to select a “bad choice”

Notation

- Relations: \( R, S \)
- Tuples: \( r, s \)
- Number of tuples: \( |R|, |S| \)
- Number of disk blocks: \( B(R), B(S) \)
- Number of memory blocks available: \( M \)
- Cost metric
  - Number of I/O’s
  - Memory requirement
- We do not count the cost of final write to disk
- Do not try to memorize the formulas for cost estimation!
  - understand the logic
  - recall the diagram of disk and memory on whiteboard
### Scanning-based algorithms

- **Table scan**
  - Scan table R and process the query
    - Selection over R
    - Projection of R without duplicate elimination
  - I/O's: \( B(R) \)
    - Trick for selection: stop early if it is a lookup by key
  - Memory requirement: 2
    - Not counting the cost of writing the result out
    - Same for any algorithm!
    - Maybe not needed—results may be pipelined into another operator

- **Nested-loop join**
  - \( R \bowtie_p S \)
    - For each block of \( R \), and for each \( r \) in the block:
      - For each block of \( S \), and for each \( s \) in the block:
        - \( R \) is called the outer table; \( S \) is called the inner table
        - I/O's: \( B(R) + |R| \cdot B(S) \)
        - Memory requirement: 3
      - Improvement: block-based nested-loop join
    - For each block of \( R \), for each block of \( S \):
      - For each \( r \) in the \( R \) block, for each \( s \) in the \( S \) block:
        - I/O's: \( B(R) + B(R) \cdot B(S) \)
        - Memory requirement: same as before

### More improvements

- Stop early if the key of the inner table is being matched
- Make use of available memory
  - Stuff memory with as much of \( R \) as possible, stream \( S \) by, and join every \( S \) tuple with all \( R \) tuples in memory
  - I/O's: \( B(R) + \frac{|R|}{M} \cdot B(S) \)
    - Or, roughly: \( B(R) \cdot \frac{B(S)}{M} \)
  - Memory requirement: \( M \) (as much as possible)
  - Which table would you pick as the outer?

### Sorting-based algorithms

- **External merge sort**
  - Remember (internal-memory) merge sort?
  - Problem: sort \( R \), but \( R \) does not fit in memory
    - **Pass 0**: read \( M \) blocks of \( R \) at a time, sort them, and write out a level-0 run
    - **Pass 1**: merge \( (M - 1) \) level-0 runs at a time, and write out a level-1 run
    - **Pass 2**: merge \( (M - 1) \) level-1 runs at a time, and write out a level-2 run
      - Final pass produces one sorted run
Toy example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 3, 6, 9
- Pass 0
  - 1, 7, 4 → 1, 4, 7
  - 5, 2, 8 → 2, 5, 8
  - 9, 6, 3 → 3, 6, 9
- Pass 1
  - 1, 4, 7 + 2, 5, 8 → 1, 2, 4, 5, 7, 8
  - 3, 6, 9
- Pass 2 (final)
  - 1, 2, 4, 5, 7, 8 + 3, 6, 9 → 1, 2, 3, 4, 5, 6, 7, 8, 9

Analysis

- Pass 0: read $M$ blocks of $R$ at a time, sort them, and write out a level-0 run
  - There are $\lfloor \log_2 M \rfloor$ level-0 sorted runs
- Pass $i$: merge $(M - 1)$ level-$(i - 1)$ runs at a time, and write out a level-$i$ run
  - $(M - 1)$ memory blocks for input, 1 to buffer output
  - # of level-$i$ runs = $\lceil \frac{\# \text{of level-} (i - 1) \text{runs}}{M - 1} \rceil$
- Final pass produces one sorted run

Performance of external merge sort

- Number of passes: $\lceil \log_{M - 1} \left( \frac{B(R)}{M} \right) \rceil + 1$
- I/O’s
  - Multiply by $2 \cdot B(R)$: each pass reads the entire relation once and writes it once
  - Subtract $B(R)$ for the final pass
  - Roughly, this is $O(B(R) \times \log_B(B(R)))$
- Memory requirement: $M$ (as much as possible)

Some tricks for sorting

- Double buffering
  - Allocate an additional block for each run
  - Overlap I/O with processing
  - Trade-off: smaller fan-in (more passes)
- Blocked I/O
  - Instead of reading/writing one disk block at time, read/write a bunch ("cluster")
  - More sequential I/O’s
  - Trade-off: larger cluster → smaller fan-in (more passes)

Sort-merge join

$R \bowtie_{R.A=S.B} S$

- Sort $R$ and $S$ by their join attributes; then merge
  - $r, s$ = the first tuples in sorted $R$ and $S$
  - Repeat until one of $R$ and $S$ is exhausted:
    - If $r.A > s.B$ then $s$ = next tuple in $S$
    - Else if $r.A < s.B$ then $r$ = next tuple in $R$
    - Else output all matching tuples, and $r, s$ = next in $R$ and $S$
- I/O’s: sorting + $2B(R) + 2B(S)$
  - In most cases (e.g., join of key and foreign key)
  - Worst case is $B(R) \cdot B(S)$: everything joins

Example of merge join

$R$:
- $r_1.A = 1$
- $r_2.A = 3$
- $r_3.A = 3$
- $r_4.A = 5$
- $r_5.A = 7$
- $r_6.A = 7$
- $r_7.A = 8$

$S$:
- $s_1.B = 1$
- $s_2.B = 2$
- $s_3.B = 3$
- $s_4.B = 3$
- $s_5.B = 8$

$R \bowtie_{R.A=S.B} S$:
- $r_1s_1$
- $r_2s_3$
- $r_2s_4$
- $r_3s_3$
- $r_3s_4$
- $r_7s_5$
Optimization of SMJ

- Idea: combine join with the (last) merge phase of merge sort
- Sort: produce sorted runs for $R$ and $S$ such that there are fewer than $M$ of them total
- Merge and join: merge the runs of $R$, merge the runs of $S$, and merge-join the result streams as they are generated!

Performance of SMJ

- If SMJ completes in two passes:
  - I/Os: $3 \cdot (B(R) + B(S))$
  - Memory requirement
    - We must have enough memory to accommodate one block from each run: $M > \frac{B(R)}{N} + \frac{B(S)}{M}$
    - $M > \sqrt{B(R) + B(S)}$
- If SMJ cannot complete in two passes:
  - Repeatedly merge to reduce the number of runs as necessary before final merge and join

Other sort-based algorithms

- Union (set), difference, intersection
  - More or less like SMJ
- Duplication elimination
  - External merge sort
    - Eliminate duplicates in sort and merge
- Grouping and aggregation
  - External merge sort, by group-by columns
    - Trick: produce "partial" aggregate values in each run, and combine them during merge
    - This trick doesn't always work though
    - Examples: SUM(DISTINCT ...), MEDIAN(...)

Hashing-based algorithms

Hash join

$R \bowtie_{R.A=S.B} S$

- Main idea
  - Partition $R$ and $S$ by hashing their join attributes, and then consider corresponding partitions of $R$ and $S$
  - If $r.A$ and $s.B$ get hashed to different partitions, they don’t join
  - Nested-loop join considers all slots
  - Hash join considers only those along the diagonal!

Partitioning phase

- Partition $R$ and $S$ according to the same hash function on their join attributes
Probing phase

• Read in each partition of $R$, stream in the corresponding partition of $S$, join
  • Typically build a hash table for the partition of $R$
    • Not the same hash function used for partition, of course: why?

<table>
<thead>
<tr>
<th>Disk</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$ partitions</td>
<td>$S$ partitions</td>
</tr>
<tr>
<td>load</td>
<td>stream</td>
</tr>
</tbody>
</table>

For each $S$ tuple, probe and join

Performance of (two-pass) hash join

• If hash join completes in two passes:
  • I/O’s: $3 \cdot (B(R) + B(S))$
  • Memory requirement:
    • In the probing phase, we should have enough memory to fit one partition of $R$: $M - 1 > \frac{B(R)}{\#1}$
    • $M > \sqrt[3]{B(R)} + 1$
    • We can always pick $R$ to be the smaller relation, so:
      $M > \min(B(R), B(S)) + 1$

End of Lecture 16