Announcements (February 17)

- Reading assignment for this week
  - Variant indexes (due Wednesday)
- Homework #1 is being graded
  - Sample solution available outside my office
- Homework #2 due February 26
- Midterm and course project proposal in 2½ weeks

Overview

- Many different ways of processing the same query
  - Scan? Sort? Hash? Use an index?
  - All with different performance characteristics
- Best choice depends on the situation
  - Implement all alternatives
  - Let the query optimizer choose at run-time
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

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 (double buffering)
- Not counting the cost of writing the result out
  - Same for any algorithm!
  - Maybe not needed—results may be pipelined directly into another operator

Nested-loop join

- $R \bowtie S$
- For each block of $R$, and for each $r$ in the block:
  - For each block of $S$, and for each $s$ in the block:
    - Output $rs$ if $p$ evaluates to true over $r$ and $s$
      - $R$ is called the outer table; $S$ is called the inner table
- I/O's: $B(R) + |R| \cdot B(S)$
- Memory requirement: 4 (double buffering)
- Improvement:
More improvements of nested-loop join

- Stop early
  - If the key of the inner table is being matched
  - May reduce half of the I/O's for unoptimized nested-loop
- Make use of available memory

External 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
  - There are $\lceil B(R)/M \rceil$ 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 = $\lceil \# \text{ of level-}($i-1$) \text{ runs} / (M - 1) \rceil$
- Final pass produces 1 sorted run

Example of external merge sort

- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3, 0
- Each block holds one number, and memory has 3 blocks
- Pass 0
  - 1, 7, 4 $\rightarrow$ 1, 4, 7
  - 5, 2, 8 $\rightarrow$ 2, 3, 8
  - 9, 6, 3 $\rightarrow$ 3, 6, 9
  - 0 $\rightarrow$ 0
- Pass 1
  - 1, 4, 7 $+$ 2, 5, 8 $\rightarrow$ 1, 2, 4, 5, 7, 8
  - 3, 6, 9 $+_{0}$ $\rightarrow$ 0, 3, 6, 9
- Pass 2 (final)
  - 1, 2, 4, 5, 7, 8 $+$ 0, 3, 6, 9 $\rightarrow$ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
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 $O(B(R) \cdot \log_M B(R))$
- Memory requirement: $M$ (as much as possible)

Some tricks for sorting

- Double buffering
  - Allocate an additional block for each run
  - Trade-off: smaller fan-in (more passes)
- Blocked I/O
  - Instead of reading/writing one disk block at time, read/write a bunch (“cluster”)
  - Trade-off: more sequential I/O’s ↔ smaller fan-in (more passes)
- Dealing with input whose size is not an exact power of fan-in

Internal sort algorithm

- Quicksort
  - Fast
- Replacement selection
  - One block for input, one for output, rest for a heap
  - Fill the heap with input records
  - Find the smallest record in the heap that is no less than the largest record in the current run
    - If that exists, move it to the output buffer, and move a new record from input buffer into the heap
    - If that does not exist, flush output and start a new run
  - Slower than quicksort, but produces longer runs (twice the size of memory if records are in random order)
Sort-merge join

- \( R \bowtie_{r.A = s.B} S \)
- Sort \( R \) and \( S \) by their join attributes, and 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 + \( 2 \ B(R) + 2 \ B(S) \)
  - In most cases (e.g., join of key and foreign key)
  - Worst case is \( B(R) \cdot B(S) \): everything joins

Example

\[
\begin{array}{ccc}
R: & S: & R \bowtie_{r.A = s.B} S: \\
\Rightarrow r_1.A = 1 & \Rightarrow s_1.B = 1 & \Rightarrow r_1.s_1 \\
\Rightarrow r_2.A = 3 & \Rightarrow s_2.B = 2 & \Rightarrow r_2.s_3 \\
\Rightarrow r_3.A = 3 & \Rightarrow s_3.B = 3 & \Rightarrow r_3.s_4 \\
\Rightarrow r_4.A = 5 & \Rightarrow s_4.B = 3 & \Rightarrow r_3.s_4 \\
\Rightarrow r_5.A = 7 & \Rightarrow s_5.B = 8 & \Rightarrow r_5.s_7 \\
\Rightarrow r_6.A = 7 & \Rightarrow & \\
\Rightarrow r_7.A = 8 & \Rightarrow & \\
\end{array}
\]

Optimization of SMJ

- Idea: combine join with the merge phase of merge sort
- Sort: produce sorted runs of size \( M \) for \( R \) and \( S \)
- 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 two-pass SMJ**

- I/Os: \( 3 \cdot (B(R) + B(S)) \)
- Memory requirement
  - To be able to merge in one pass, we should have enough memory to accommodate one block from each run: \( M > B(R) / M + B(S) / M \)
  - \( M > \sqrt[3]{B(R) + B(S)} \)

**Other sort-based algorithms**

- Union (set), difference, intersection
  - More or less like SMJ
- Duplication elimination
  - External merge sort
    - Eliminate duplicates in sort and merge
- **GROUP BY** and aggregation
  - External merge sort
    - Produce partial aggregate values in each run
    - Combine partial aggregate values during merge
    - Partial aggregate values don’t always work though
      - Examples: SUM(DISTINCT …), MEDIAN(…)

**Hash join**

- \( R \bowtie_{R,A} S \)
- 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!

Performance of hash join

- 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 $R$: $M - 1 \geq B(R)/(M - 1)$
  - $M > \sqrt{B(R)}$
  - We can always pick $R$ to be the smaller relation, so: $M > \sqrt{\min(B(R), B(S))}$
Hash join tricks

- What if a partition is too large for memory?
  - Read it back in and partition it further!
  - See the duality in multi-pass merge sort here?

Hybrid hash join

- What if there is extra memory available?
  - Use it to avoid writing/re-reading partitions
  - Of both \( R \) and \( S \)!

A generalization of the idea is described in the survey paper by Graefe

Hash join versus SMJ

(Assuming two-pass)

- I/O’s: same
- Memory requirement: hash join is lower
  - \( \sqrt{\text{min}(\text{B}(R), \text{B}(S))} < \sqrt{\text{B}(R) + \text{B}(S)} \)
  - Hash join wins big when two relations have very different sizes
- Other factors
What about nested-loop join?

Other hash-based algorithms

- Union (set), difference, intersection
  - More or less like hash join
- Duplicate elimination
  - Check for duplicates within each partition/bucket
- GROUP BY and aggregation
  - Apply the hash functions to GROUP BY attributes
  - Tuples in the same group must end up in the same partition/bucket
  - Keep a running aggregate value for each group

Duality of sort and hash

- Divide-and-conquer paradigm
  - Sorting: physical division, logical combination
  - Hashing: logical division, physical combination
- Handling very large inputs
  - Sorting: multi-level merge
  - Hashing: recursive partitioning
- I/O patterns
  - Sorting: sequential write, random read (merge)
  - Hashing: random write, sequential read (partition)