Query Processing

CompSci 316
Introduction to Database Systems

Announcements (Tue. Nov. 13)
- Project Milestone #2 due Thursday
- Homework #4 will be assigned Thursday
- My office hours today are moved to Wed. 2-3pm

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
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 (+1 for double buffering)
- 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:
    - 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: 3 (+1 for double buffering)
  - Improvement: block-based nested-loop join
  - I/O’s: $B(R) + B(R) \cdot B(S)$
  - Memory requirement: same as before
More improvements of nested-loop join

- 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: $B(R) + \frac{B(S)}{M} \cdot B(S)$
  - Or, roughly: $B(R) \cdot B(S)/M$
- Memory requirement: M (as much as possible)
- Which table would you pick as the outer?

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

Example of external merge sort

- 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
Performance of external merge sort

- Number of passes: \( \lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \rceil + 1 \)
- 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 \( O(B(R) \times \log MB(R)) \)
- 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:
- Blocked I/O
  - Instead of reading/writing one disk block at time, read/write a bunch ("cluster")

Sort-merge join

- \( R \bowtie_{RA=S.B} S \)
- 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

\[
\begin{align*}
R: & S: & R \bowtie_{R.A=S.B} S: \\
\Rightarrow r_1.A = 1 & \Rightarrow s_1.B = 1 & r_1s_1 \\
\Rightarrow r_2.A = 3 & \Rightarrow s_2.B = 2 & r_2s_3 \\
\Rightarrow r_3.A = 3 & \Rightarrow s_3.B = 3 & r_2s_4 \\
\Rightarrow r_4.A = 5 & \Rightarrow s_4.B = 3 & r_3s_3 \\
\Rightarrow r_5.A = 7 & \Rightarrow s_5.B = 8 & r_3s_4 \\
\Rightarrow r_6.A = 7 & \Rightarrow s_6.B = 8 & r_7s_5 \\
\Rightarrow r_7.A = 8 & & \\
\end{align*}
\]

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/O’s: \( 3 \cdot (B(R) + B(S)) \)
- Memory requirement
  - To be able to merge in one pass, we should have enough memory to accommodate one block from each run:
    \[
    M > \frac{B(R) + B(S)}{M} \]
  - \( M > \sqrt{B(R) + B(S)} \)
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
    - Trick: produce partial aggregate values in each run, and combine them during merge
      - Partial aggregate values don’t always work though

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

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 $R$: $M - 1 > \frac{B(R)}{M-1}$
  - $M > \sqrt[2]{B(R)} + 1$
  - We can always pick $R$ to be the smaller relation, so:
    $$M > \sqrt{\min(B(R), B(S))} + 1$$

Hash join tricks

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

(Assuming two-pass)

- I/O’s: same
- Memory requirement: hash join is lower

\[
\sqrt{\min(B(R), B(S))} + 1 < \sqrt{B(R) + B(S)}
\]

- Hash join wins when two relations have very different sizes

Other factors

- Hash join performance depends on the quality of the hash
  - Might not get evenly sized buckets
- SMJ can be adapted for inequality join predicates
- SMJ wins if \( R \) and/or \( S \) are already sorted
- SMJ wins if the result needs to be in sorted order

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
- \texttt{GROUP BY} and aggregation
  - Apply the hash functions to \texttt{GROUP BY} attributes
  - Tuples in the same group must end up in the same partition/bucket
  - Keep a running aggregate value for each group
    - May not always work
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)

Selection using index

- Equality predicate: $\sigma_{A=p}(R)$
  - Use an ISAM, B+-tree, or hash index on $R(A)$
- Range predicate: $\sigma_{A>0}(R)$
  - Use an ordered index (e.g., ISAM or B+-tree) on $R(A)$
  - Hash index is not applicable
- Indexes other than those on $R(A)$ may be useful
  - Example: B+-tree index on $R(A, B)$
  - How about B+-tree index on $R(B, A)$?

Index versus table scan

- Situations where index clearly wins:
  - Index-only queries which do not require retrieving actual tuples
    - Example: $\pi_A(\sigma_{A=p}(R))$
  - Primary index clustered according to search key
    - One lookup leads to all result tuples in their entirety
Index versus table scan (cont’d)

BUT(!):

- Consider $\sigma_{A>v}(R)$ and a secondary, non-clustered index on $R(A)$
  - Need to follow pointers to get the actual result tuples
  - Say that 20% of $R$ satisfies $A > v$
    - Could happen even for equality predicates
  - I/O’s for index-based selection: lookup + 20% $|R|$
  - I/O’s for scan-based selection: $B(R)$
  - Table scan wins if a block contains more than 5 tuples

Index nested-loop join

- $R \bowtie_{R.A=S.B} S$
- Idea: use a value of $R.A$ to probe the index on $S(B)$
- For each block of $R$, and for each $r$ in the block:
  - Use the index on $S(B)$ to retrieve $s$ with $s.B = r.A$
  - Output $rs$
- I/O’s: $B(R) + |R| \cdot \text{index lookup}$
  - Typically, the cost of an index lookup is 2-4 I/O’s
  - Beats other join methods if $|R|$ is not too big
  - Better pick $R$ to be the smaller relation
- Memory requirement: 3

Zig-zag join using ordered indexes

- $R \bowtie_{R.A=S.B} S$
- Idea: use the ordering provided by the indexes on $R(A)$ and $S(B)$ to eliminate the sorting step of sort-merge join
- Trick: use the larger key to probe the other index
  - Possibly skipping many keys that don’t match

\[ \begin{align*}
& \Rightarrow 1 \quad \Rightarrow 2 \\
& \Rightarrow 3 \quad \Rightarrow 4 \quad \Rightarrow 7 \quad \Rightarrow 9 \quad \Rightarrow 18 \\
& \Rightarrow 1 \quad \Rightarrow 7 \quad \Rightarrow 9 \quad \Rightarrow 11 \quad \Rightarrow 12 \quad \Rightarrow 17 \quad \Rightarrow 19
\end{align*} \]
Summary of tricks

- **Scan**
  - Selection, duplicate-preserving projection, nested-loop join

- **Sort**
  - External merge sort, sort-merge join, union (set), difference, intersection, duplicate elimination, GROUP BY and aggregation

- **Hash**
  - Hash join, union (set), difference, intersection, duplicate elimination, GROUP BY and aggregation

- **Index**
  - Selection, index nested-loop join, zig-zag join