Query Processing
Introduction to Databases
CompSci 316 Spring 2020

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

Scanning-based algorithms
• 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

Recall our disk-memory diagram
On board!
• How do we implement \textit{Join}?
• Ideas? (discuss with neighbors)
• Cost?
  • (page I/O – in terms of B(R), |R| etc.)
  • Memory requirement?

\textbf{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/Os: $B(R) + |R| \cdot B(S)$
• Memory requirement: \( \leq B(R) \)

\textbf{Block-based Nested Loop Join}

\[ R \bowtie_p S \]

• $R$ outer, $S$ inner
• 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/Os: $B(R) + B(R) \cdot B(S)$
  • Memory requirement: same as before

\textbf{More improvements}

• Make use of available memory
  • Stuff memory with as much of $R$ as possible, stream $S$
  • I/Os: $B(R) + \frac{B(R)}{M} \cdot B(S)$
    • Or, roughly: $B(R) \cdot B(S) / M$
  • Memory requirement: \( \leq M \) (as much as possible)

• Which table would you pick as the outer?

\textbf{Sorting-based algorithms}

\[ \text{http://en.wikipedia.org/wiki/Mail_sorter#mediaviewer/File:Mail_sorting,1951.jpg} \]

\textbf{External merge sort}

Remember (internal-memory) merge sort?
Problem: sort $R$, but $R$ does not fit in memory
• \textbf{Pass 0}: read $M$ blocks of $R$ at a time, sort them, and write out a level-0 run

• \textbf{Pass 1}: merge $(M - 1)$ level-0 runs at a time, and write out a level-1 run

• \textbf{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
- 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 $\frac{(B(R))}{M}$ 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 $= \frac{\text{# of level-$(i-1)$ runs}}{M-1}$
- Final pass produces one sorted run

Performance of external merge sort

- Number of passes: $\log_{M-1} \left( \frac{B(R)}{M} \right) + 1$
- 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 $O(B(R) \times \log_{M} B(R))$
- 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 $r.A \geq s.B$ then $s = \text{next tuple in } S$
    - If $r.A < s.B$ then $r = \text{next tuple in } R$
  - else output all matching tuples, and $r, s = \text{next in } R$ and $S$
- I/O’s: $\text{sorting} + 2B(R) + 2B(S)$ (always?)
  - In many cases (e.g., join of key and foreign key)
  - Worst case is $B(R) \cdot B(S)$: everything joins

Example of merge join

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

<table>
<thead>
<tr>
<th>$r_1$</th>
<th>$A = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_2$</td>
<td>$A = 3$</td>
</tr>
<tr>
<td>$r_3$</td>
<td>$A = 3$</td>
</tr>
<tr>
<td>$r_4$</td>
<td>$A = 5$</td>
</tr>
<tr>
<td>$r_5$</td>
<td>$A = 7$</td>
</tr>
<tr>
<td>$r_6$</td>
<td>$A = 7$</td>
</tr>
<tr>
<td>$r_7$</td>
<td>$A = 8$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$B = 1$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$B = 2$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$B = 3$</td>
</tr>
<tr>
<td>$s_4$</td>
<td>$B = 3$</td>
</tr>
<tr>
<td>$s_5$</td>
<td>$B = 8$</td>
</tr>
</tbody>
</table>

| $r_1s_1$ |
| $r_2s_3$ |
| $r_3s_4$ |
| $r_4s_3$ |
| $r_5s_4$ |
| $r_6s_5$ |
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)) - why 3?
  - Memory requirement:
    - We must have enough memory to accommodate one block from each run: \( M > \frac{B(R) + B(S)}{M - 1} \)
    - \( M > \sqrt{B(R) + B(S)} \)
- 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

\[ R \bowtie_{A=S.B} S \]

- 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!

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

Generalizing for larger inputs

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

• May be best if many tuples join
  • Example: non-equality joins that are not very selective

• Necessary for black-box predicates
  • Example: WHERE user_defined_pred($R.A, S.B$)

Other hash-based algorithms

• Union (set), difference, intersection
  • More or less like hash join
• Duplicate elimination
  • Check for duplicates within each partition/bucket
• Grouping and aggregation
  • Apply the hash functions to the group-by columns
    • 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: $\sigma_{A=v}(R)$
  - Use an ISAM, B+-tree, or hash index on $R(A)$
- Range predicate: $\sigma_{A>v}(R)$
  - 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: $\pi_A(\sigma_{A>v}(R))$
- 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
- Use the larger key to probe the other index
  - Possibly skipping many keys that don't match

Summary of techniques

- Scan
  - Selection, duplicate-preserving projection, nested-loop join
- Sort
  - External merge sort, sort-merge join, union (set), difference, intersection, duplicate elimination, grouping and aggregation
- Hash
  - Hash join, union (set), difference, intersection, duplicate elimination, grouping and aggregation
- Index
  - Selection, index nested-loop join, zig-zag join