Query Processing

CPS 216
Advanced Database Systems

Overview

- Many different ways of implementing the same logical query operator
  - Scan, sort, hash, 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 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 (double buffering)

Tricks for nested-loop join

- Stop early
  - If the key of the inner table is being matched
  - May reduce half of the I/O’s
- Block-based nested-loop join
  - 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: $B(R) + \lceil B(R) / (M - 2) \rceil \cdot B(S)$
    - Or, roughly: $B(R) \cdot B(S) / M$
  - Memory requirement: $M$ (as much as possible)
**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
- **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:** \( \lceil \log_{M-1} \left( \frac{B(R)}{M} \right) \rceil + 1 \)
- **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 \( O(B(R) \cdot \log B(R)) \)
- **Memory requirement:** \( M \) (as much as possible)

**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”)
  - More sequential I/O’s
  - Trade-off: larger cluster ↔ smaller fan-in (more passes)
- **Replacement sort**
  - On average produces level-0 runs that are twice as big
  - Use a priority heap: keep outputting as much as possible and making space for input

**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 + \( B(R) + B(S) \)
  - In most cases (e.g., join of key and foreign key)
  - Worst case is \( B(R) \cdot B(S) \): everything joins

**Example**

- **R:**
  - \( r_1.A = 1 \)  \( s_1.B = 1 \)  \( r_1.s_1 \)
  - \( r_2.A = 3 \)  \( s_1.B = 2 \)  \( r_2.s_3 \)
  - \( r_3.A = 5 \)  \( s_2.B = 3 \)  \( r_3.s_4 \)
  - \( r_4.A = 7 \)  \( s_2.B = 8 \)  \( r_3.s_5 \)
  - \( r_5.A = 7 \)  \( r_5.s_5 \)
- **S:**
  - \( r_2.A = 8 \)
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 the result streams as they are generated!

Other sort-based algorithms

- Union, 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

- $R \bowtie_{r.A = s.B} S$
- 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 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: $M > B(R)/M + B(S)/M$
  - $M > \sqrt{B(R) + B(S)}$
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: $M > \sqrt{\text{min}(B(R), B(S))}$

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?

Hybrid hash join
• What if there is extra memory available?
  – Use it to avoid writing/re-reading partitions
    • Of both $R$ and $S$!

Hash join versus SMJ
(Assuming two-pass)
• I/O’s: same
• Memory requirement: hash join is lower
  – $\sqrt{\text{min}(B(R), B(S))} < \sqrt{B(R) + B(S)}$
  – 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, 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)