Lecture 4: External sorting, Evaluation of relational operators

Rasmus Pagh

Today's lecture

- Morning session: External sorting
  - Motivation, recap of merge-sort
  - Buffer management
  - Analysis and external memory version
  - Lower bound
- Afternoon session: Relational operators
  - Several algorithms for select, join, grouping,...
  - Analysis of the algorithms
  - Comparison of algorithms
  - Exercises
- 14.55 PM: Minister Helge Sander in Aud. 1

Why study sorting?

- To prepare for your job interview at Google?

  http://www.youtube.com/watch?v=k4RRi_ntQc8

Why study sorting?

1. Basis for many efficient algorithms, especially in blocked memory.
2. Reminds us that massive data is a different world:
    1. Bucket sorting may be worse than superlinear algorithms.
3. More practice in analyzing the performance of external memory algorithms.

- Recap: Merge sort (board).

Analysis of disk-based algorithms

Two worlds:

- External memory algorithmics:
  - The algorithm decides when to read and write blocks (pages).
- DBMSs (and operating systems):
  - A *buffer manager* decides what pages are kept in memory.
  - Sometimes the buffer manager may be forced to write a page to disk.
  - Algorithms may prioritize data (memory is split into *buffer pools*).

Buffer management in a nutshell

- Keep track of pages that are currently being accessed (pinning).
- Keep track of pages that have changed since they were read (dirty).
- Perhaps try to predict future reads (prefetching).
- When space is needed in a buffer pool, use a *replacement policy* to determine which page to remove from memory.
  - LRU, clock, FIFO, MRU, random,... (board)

Analysis of merge sort with LRU

- Notation: (different from RG)
  - N data items, B in each block.
  - M data items fit in internal memory.
- In each of $\log_2 N$ phases, all data items are read and written once.

$\Rightarrow$ At most $2(N/B)\log_2 N$ I/Os.

- But mergesort has *locality of reference*: *Runs* of M items are sorted completely without accesing other items.

$\Rightarrow$ At most $2(N/B)\log_2(2N/M)$ I/Os.

Improving merge sort 1/3

Forming longer runs in one scan:

1. `Fill memory with M data items.`
2. `Let maxOutput =` $- \infty$.
3. `REPEAT`
   a) `Output a sorted block containing the smallest B items in memory ≥ maxOutput.`
   b) `Let maxOutput:=the largest item written.`
   c) `Read a new block into memory.`
4. `UNTIL at most B items ≥ maxOutput.`
5. `Goto 1 (use items already read).`

On "average", runs are twice as long.

Improving merge sort 2/3

- Make better use of internal memory:

  Merge M/B runs instead of 2 (board)

- Number of phases drops:
  - One phase for forming runs
  - $\log_{M/B}(N/M)$ phases for merging
- In almost all settings: 1 merge phase!
- Number of I/Os per phase is 2N/B. (Sometimes, final write is not needed.)

Improving merge sort 3/3

- The disk access pattern is predictable, to some extent. Use in two ways:

1. **Blocked I/O**: Always read a number of consecutive disk blocks.
2. **Double buffering**: Ask for new blocks *before* they are needed.

- The price for both techniques is a higher internal memory requirement.

External merge sort summary

- M = internal memory usage.
- In first phase, can sort runs of M (or sometimes 2M) elements.
- In second phase, can merge (up to) M/B runs into a sorted run of (up to) M^2/B elements.
- Sorting is completed in two phases if $N<M^2/B$, or equivalently if $M>(NB)^{1/2}$.

Improving even more?

- The method you have seen is known to be essentially optimal.
- Don't believe claims of products that do much better (except in special cases).

- Board presentation:
  - Lower bound for comparison-based sorting (recap).
  - Lower bound for comparison-based external memory sorting.

Relational algebra operations

- Relational DBMSs compute query results by performing a sequence of relational algebra operations:
  - Selections (σ)
  - Projections (π)
  - Joins ($\bowtie$)
  - Groupings and aggregations (γ)
  - Set operations ($\cup$,$\cap$,-)
  - Duplicate elimination (δ)
- Today, we focus on how to perform each single operation.

Selection

- We focus on the conjunction (”and”) of a number of equality and range conditions.
- Two main cases:
  - No relevant index. (What is that?)
    In this case, a full table scan is required.
  - One or more relevant indexes.
    a) There is a highly selective condition with a matching index.
    b) No single condition matching an index is highly selective.

Using a highly selective index

- Basic idea:
  - Retrieve all matching tuples (few)
  - Filter according to remaining conditions

- If index is clustered, retrieving matching tuples is very efficient.
- If index is unclustered, it may be advantageous to:
  1. Retrieve pointers (RIDs) to matching tuples.
  2. Retrieve tuples order of sorted RID.

Using several less selective indexes

- For several conditions C_1, C_2,... matched by indexes:
  - Retrieve the RIDs R_i of tuples matching C_i.
  - Compute the intersection $R = R_1 \cap R_2 \cap \ldots$
  - Retrieve the tuples in R (in sorted order)

- Remaining problem:
  - How can we estimate the selectivity of a condition? Of a combination of conditions?
  - More on this next time.

Operations that require grouping

- Many operations are easy to perform once the involved tuples (in one or more relations) are groups according to the values of some attributes:
  - Projections (group by output attributes)
  - Join with equality condition (group by join attributes)
  - Groupings and aggregations (obvious)
  - Set operations (group by all attributes)
  - Duplicate elimination (group by all attributes)

Two principles for grouping

- Sorting the tuples
  - All the mentioned operations can be performed during the merge phase, i.e., no need to materialize the sorted list.
  - (For join this is not clear when there are many duplicates - exercise later today.)
- Hashing the tuples
  - Hash to as many buckets as memory allows (need one output buffer for each bucket).
  - If each bucket fits in memory, grouping can be done by reading each bucket.

Pros and cons

- Sorting-based grouping is *deterministic*, i.e., no chance of bad behaviour.
- Sorting-based grouping outputs the result in sorted order
  - For union, intersection, and projection we may freely choose the order.
- Next: Hashing-based grouping uses less memory for joins.

Problem session

- Consider an equality-join of relations R_1 and R_2.
  - R_1 uses $B(R_1)$ blocks of disk space.
  - R_2 uses $B(R_2) > B(R_1)$ blocks of disk space.
- Suppose that hashing distributes keys in a completely uniform way.
- Argue that 2 passes suffices to do a hash-based join if $B(R_1) < M^2/B$. (Memory independent of $B(R_2)$!)

Sometimes simple suffices

- An alternative way of doing joins (with *general conditions*) is a *block nested loop join*.
- Simple idea:
  - Divide R_1 into blocks of size M.
  - For each block:
    a) Read the block into memory
    b) Scan R_2 for matching tuples
- Complexity is $B(R_2)B(R_1)/(M/B)$.
  - Good if R_1 (almost) fits in internal memory.

Hybrid hash join

- Goal: Always get the best of hash join and block nested loop join.
- Idea:
  - First partition the smaller relation, but...
  - Do not write all partitions to disk, keep as much data as possible in internal memory
  - When partitioning the larger relation, check for matching tuples in memory.

Index nested loop join

- If there is an index that matches the join condition, the following algorithm can be considered:
- For each tuple in R_1, use the index to locate matching tuples in R_2.
- In general, the cost is at least 1 I/O (hash index) for each tuple in R_1.
  - Use only if $|R_1|$ is small compared to $B(R_2)$.
  - (Why are we interested in small relations?)
- If many tuples may match each tuple, a clustered index is preferable.

Summary

- Several algorithms possible for each operator:
  - Use index or not (selection, join)?
  - Use several indexes (selection)?
  - Sort- or hash-based?
- Choosing the best is complicated in general - more on this next time.

Exercises

- Robust sort-merge join
- Sort-merge-based join on partly sorted input