

# Lecture 5: Evaluation of relational operators

Rasmus Pagh

# Today's lecture

- Motivating case study.
- External sorting
  - Motivation, recap of merge-sort
  - Buffer management
  - Analysis and external memory version
  - Lower bound
- Relational operators
  - Several algorithms for select, join, grouping,...
  - Analysis of the algorithms
  - Comparison of algorithms



# Case study

- Relation StarsIn(movie, actor).  
Sorted by actor name.
- Find all co-actors:  

```
SELECT DISTINCT s1.name, s2.name  
FROM StarsIn s1, StarsIn s2  
WHERE s1.movie=s2.movie
```
- Assume:
  - a clustered index on (movie,actor) exists,
  - there are 10 million rows in StarsIn, and
  - 1 million movies (similar in size to IMDB).

# 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.



## Back to case study

- Index nested loop join will create 10 million index lookups:
  - 100,000 seconds at 100 lookups/second
  - And this does not include the time for eliminating duplicates...
- *Buffer manager* helps a bit.



# 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)

## Case study, again

- As a simple test, I ran this query on a subset of 500,000 tuples of starsIn (8,000 movies) from IMDB.
- MySQL used an index on movie to perform the join.
- Stopped without finishing the query after 7 hours...

# Why study sorting?

1. Basis for many efficient algorithms, especially in blocked memory.
2. Reminds us that massive data is a different world:
  - 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 algorithms:
  - The algorithm decides when to read and write blocks (pages).
- DBMSs (and operating systems):
  - *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*).



# 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.  
⇒ At most  $2(N/B)\log_2 N$  I/Os.
- But mergesort has *locality of reference*:  
*Runs* of M items are sorted completely without accessing other items.  
⇒ 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  $\text{maxOutput} = -\infty$ .
3. REPEAT
  - a) Output a sorted block containing the smallest B items in memory  $\geq \text{maxOutput}$ .
  - b) Let  $\text{maxOutput} :=$  the largest item written.
  - c) Read a new block into memory.
4. UNTIL at most B items  $\geq \text{maxOutput}$ .
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  $M > (NB)^{1/2}$ .



# Sorting, concretely

- Assume two passes:  
All data read and written once.
- Transfer rate for a typical disk drive:  
60 MB/sec.
- Assume I/O is bottleneck (a realistic assumption).
  
- Conclusion:  
Can sort 15 MB of data per second.

# Improving even more?

- The method you have seen is known to be essentially optimal:
  - It can be shown that **any** sorting algorithm uses (nearly) the same number of I/Os as multi-way mergesort.
- **Idea:** Consider the tree of all possible behaviors of an algorithm:
  - Must have  $N!$  leaves to sort.
  - Depth corresponds to number of I/Os.



# Relational algebra operations

- Recall: Relational DBMSs compute query results by performing a sequence of relational algebra operations:
  - Selections ( $\sigma$ )
  - Projections ( $\pi$ )
  - Joins ( $\bowtie$ )
  - Groupings and aggregations ( $\gamma$ )
  - Set operations ( $\cup, \cap, -$ )
  - Duplicate elimination ( $\delta$ )
- Next: 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.  
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  $C_1, C_2, \dots$  matched by indexes:
  - Retrieve the RIDs  $R_i$  of tuples matching  $C_i$ .
  - Compute the intersection  $R = R_1 \cap R_2 \cap \dots$
  - 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 grouped according to the values of some attribute(s):
  - 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.
  - If two-phase:  $3(N/B)$  I/Os (excl. output).
- 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.



## Case study, again

- Suppose that instead of index nested loop join, a sort-based join is used.
  - 600 MB of data, takes around 30 seconds to produce join result (excluding output).
- Suppose intermediate result has size 6000 MB, and final result 600 MB.
  - Duplicate elimination requires one write and one read of all data (excl. output) because of pipelining.
  - 200 seconds to generate result, 10 seconds to output.
- Total time: 4 minutes (vs 24+ hours)



# Sorting vs hashing

- 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 suffice 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.



# 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 result size estimation next time.

