Lecture #11: Sorting & Aggregation Algorithms

15-445/645 Database Systems (Fall 2025)
`https://15445.courses.cs.cmu.edu/fall2025/`
Carnegie Mellon University
Andy Pavlo

1 Query Plan

Up to this point we have talked about access methods. Now we need to actually execute queries.

The database system will compile SQL into a query plan, which is a tree or DAG of operators. We will cover more details about operators etc. in later lectures.

For a disk-oriented database system, we will use the buffer pool to implement algorithms that need to spill to disk. We want to minimize I/O for an algorithm and prefer sequential over random I/O.

2 Sorting

DBMSs need to sort because tuples in a table have no specific order under the relational model, so `ORDER BY` requires it. Also, operators like `GROUP BY`, `JOIN`, and `DISTINCT` potentially use sorting. If the data that needs to be sorted fits in memory, then the DBMS can use a standard sorting algorithm (e.g., quicksort, vergesort). If the data does not fit, then the DBMS needs to use external sorting that keeps in mind the cost of disk operations. The sorting algorithm sorts a given input *run*, that is a list of key-value pairs, and sort it based on comparison function and sorting parameters. The key is the attributes to compare to compute the sort order, and the value could be the whole tuple (early materialization) or the record ID only (late materialization).

Top-N Heap Sort

If a query contains an `ORDER BY` with a `LIMIT` and an optional `WITH TIES`, then to find the top-N elements, the DBMS only needs to scan the data once. During the scan, the DBMS can maintain a priority queue storing the top-N elements seen so far. If the query contains `WITH TIES`, the priority queue would be extended on-demand to keep the ties. The ideal scenario is when the top-N elements fit in memory, so the DBMS can maintain the entire priority queue in-memory.

External Merge Sort

This is the standard algorithm for sorting data which is too large to fit in memory. It is a divide-and-conquer sorting algorithm that splits the data set into separate *runs* and then sorts them individually. It can spill runs to disk as needed then read them back in one at a time. The algorithm is comprised of two phases:

- **Phase #1 – Sorting:** First, the algorithm sorts small chunks of data that fit in main memory, and then writes the sorted pages back to disk.
- **Phase #2 – Merge:** Then, the algorithm combines the sorted runs into larger sorted runs.

A sorted run can be early-materialized, which means that the entire tuple is stored in the pages, or can be late-materialized, which means we only store record IDs in the pages and read them later.

Two-way Merge Sort: The most basic version of the external merge sort algorithm is the two-way merge sort. During the sorting phase, the algorithm reads each page, sorts it, and writes the sorted version back to disk. Then, in the merge phase, it uses three buffer pages. It reads two sorted pages in from disk, and merges them together into a third buffer page. Whenever the third page fills up, it is written back to disk and replaced with an empty page. Each set of sorted pages is called a *run*. The algorithm then recursively merges the runs together.

Let N be the total number of data pages. The algorithm makes $1 + \lceil log_2 N \rceil$ total passes through the data (1 for the first sorting step then $\lceil log_2 N \rceil$ for the recursive merging). The total I/O cost is $2N \times$ (# of passes) since each pass performs a read and a write for each page.

General (K-way) Merge Sort: The generalized version of the algorithm allows the DBMS to take advantage of more than three buffer pages. Let B be the total number of buffer pages available. Then, during the sort phase, the algorithm can read and sort B pages at a time, so the DBMS writes $\lceil \frac{N}{B} \rceil$ sorted runs of length B pages back to disk. The merge phase can combine up to $B - 1$ runs in each pass, using $B - 1$ buffer pages to step through the runs and using one buffer page for the combined data, writing back to disk as needed.

In the generalized version, the algorithm performs $1 + \lceil log_{B-1} \lceil \frac{N}{B} \rceil \rceil$ passes (one for the sorting phase and $\lceil log_{B-1} \lceil \frac{N}{B} \rceil \rceil$ for the merge phase. Then, the total I/O cost is $2N \times$ (# of passes) since it again has to make a read and write for each page in each pass.

Double Buffering Optimization: One optimization for external merge sort is prefetching the next run in the background and storing it in a second set of buffers while the system is processing the current run. This reduces the wait time for I/O requests at each step by continuously utilizing the disk. However, it reduces effective buffers available by half. This optimization requires the use of multiple threads, since the prefetching should occur while the computation for the current run is happening.

Comparison Optimizations: Comparing keys in the sorting algorithm can be expensive. DBMSs can implement different optimizations to speed up this process:

- **Code Specialization:** Instead of providing the comparator as a function pointer to the sorting algorithm, the sorting function can be hard-coded to the specific key type. An example of this is template specialization in C++.
- **Suffix Truncation:** Another optimization for string-based comparisons is only compare part of keys first. This first compares binary prefixes of long `VARCHAR` keys, falling back to an entire string comparison if the prefixes are equal.
- **Key Normalization:** This converts variable-length keys into a single encoded / padded fixed-length string that preserves sort order.

Using B+Trees

It is sometimes advantageous for the DBMS to use an existing B+tree index to aid in sorting rather than using the external merge sort algorithm. Some DBMSs support prefix key scans for sorting. In particular, if the index is a clustered index, the DBMS can just traverse the B+tree. Since the index is clustered, the I/O access will be sequential. This means it is always better than external merge sort since no computation is required. On the other hand, if the index is unclustered, traversing the tree is almost always worse, since each record could be stored in any page, so nearly all record accesses will require a disk read. The only exception might be Top-N queries, when N is small relative to the total number of tuples in a table.

3 Aggregations

An aggregation operator in a query plan collapses the values of one or more tuples into a single scalar value. There are two approaches for implementing an aggregation: (1) sorting and (2) hashing.

Sorting

The DBMS first sorts the tuples on the `GROUP BY` key(s). It can use either an in-memory sorting algorithm if everything fits in the buffer pool (e.g., quicksort) or the external merge sort algorithm if the size of the data exceeds memory. The DBMS then performs a sequential scan over the sorted data to compute the aggregation. The output of the operator will be sorted on the keys.

When performing sorting aggregations, it is important to order the query operations to maximize efficiency. For example, if the query requires a filter, it is better to perform the filter first and then sort the filtered data to reduce the amount of data that needs to be sorted.

Hashing

Hashing can be computationally cheaper than sorting for computing aggregations, especially when the order of the output does not matter. If the hash table fits in memory, build an ephemeral hash table while scanning the table. For each record, check whether there is already an entry in the hash table and perform the appropriate modification. If the hash table is too large, then the DBMS has to spill it to disk and do *External Hashing Aggregate*. This is a divide-and-conquer approach to compute the aggregation, with two phases:

- **Phase #1 – Partition:** Use a hash function h_1 to split tuples into partitions on disk based on the target hash key. This will put tuples that must be compared for the aggregation into the same partition. Assume B buffer pages in total. We will have B-1 output buffer pages for partitions and 1 buffer page for input data. If any partition is full, the DBMS will spill it to disk. Thus, this phase results in $B - 1$ partitions.
- **Phase #2 – ReHash:** For each partition on disk, read its pages into memory and build an in-memory hash table based on a second hash function h_2 (where $h_1 \neq h_2$). This will put all tuples that match into the same bucket. Then go through each bucket of this hash table to bring together matching tuples to compute the aggregation. This assumes that each partition fits in memory, which can be achieved by recursively partitioning until this is true.

During the ReHash phase, the DBMS can store pairs of the form (`GroupByKey`→`RunningValue`) to compute the aggregation. The contents of `RunningValue` depends on the aggregation function (e.g. `(COUNT, SUM)` for `AVG`). To insert a new tuple into the hash table:

- If it finds a matching `GroupByKey`, then update the `RunningValue` appropriately.
- Else insert a new (`GroupByKey`→`RunningValue`) pair.

The choice between sorting and hashing aggregate is subtle and depends on optimizations done in each case, but in general hashing is often more efficient unless the data is already sorted beforehand or the output is required to be sorted (e.g. following `ORDER BY`).