Lecture #10: Sorting & Aggregation Algorithms

15-445/645 Database Systems (Fall 2019)
`https://15445.courses.cs.cmu.edu/fall2019/`
Carnegie Mellon University
Prof. Andy Pavlo

1 Sorting

We need sorting because in the relation model, tuples in a table have no specific order Sorting is (potentially) used in `ORDER BY`, `GROUP BY`, `JOIN`, and `DISTINCT` operators.

We can accelerate sorting using a *clustered* B+tree by scanning the leaf nodes from left to right. This is a bad idea, however, if we use an *unclustered* B+tree to sort because it causes a lot of I/O reads (random access through pointer chasing).

If the data that we need to sort fits in memory, then the DBMS can use a standard sorting algorithms (e.g., quicksort). If the data does not fit, then the DBMS needs to use external sorting that is able to spill to disk as needed and prefers sequential over random I/O.

2 External Merge Sort

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.

Phase #1 – Sorting: Sort small chunks of data that fit in main memory, and then write back to disk.

Phase #2 – Merge: Combine sorted sub-files into a larger single file.

Two-way Merge Sort

1. Pass #0: Reads every B pages of the table into memory. Sorts them, and writes them back into disk. Each sorted set of pages is called a **run**.
2. Pass #1,2,3...: Recursively merges pairs of runs into runs twice as long.

Number of Passes: $1 + \lceil log_2 N \rceil$
Total I/O Cost: $2N \times$ (# of passes)

General (K-way) Merge Sort

1. Pass #0: Use B buffer pages, produce N/B sorted runs of size B.
2. Pass #1,2,3...: Recursively merge $B-1$ runs.

Number of Passes = $1 + \lceil log_{B-1} \lceil \frac{N}{B} \rceil \rceil$
Total I/O Cost: $2N \times$ (# of passes)

Double Buffering Optimization

Prefetch the next run in the background and store it in a second buffer 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.

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.

Hashing

Hashing can be computationally cheaper than sorting for computing aggregations. The DBMS populates an ephemeral hash table as it scans the table. For each record, check whether there is already an entry in the hash table and perform the appropriate modification.

If the size of the hash table is too large to fit in memory, then the DBMS has to spill it to disk:

- **Phase #1 – Partition:** Use a hash function h_1 to split tuples into partitions on disk based on target hash key. This will put all tuples that match into the same partition. The DBMS spills partitions to disk via output buffers.
- **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 $h_1 \neq h_2$). Then go through each bucket of this hash table to bring together matching tuples to compute the aggregation. Note that this assumes that each partition fits in memory.

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. 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.