CS 448 Course Notes

Database Systems Implementation

Michael Socha

University of Waterloo
Winter 2019

Contents

1 Course Overview

This is a course about the inner workings of database management systems (DBMS). Key topics covered include:

- Storage systems and disk-based data structures
- Query processing and optimization
- Transactions

2 Disks and Files

2.1 Data Storage Devices

Most data in databases is stored on hard disks, which have their memory divided into pages. A random page can be retrieved at some fixed cost. Tapes can be used as a cheaper alternative to disks, but their data must be read in order.

2.1.1 Accessing Disk Page

Time to access (i.e. read/write) a disk block can be estimated by summing the following:

- **Seek time**, which is the times it take to move the disk arms to the head of the disk block.
- **Rotational delay**, which is the time it takes for the disk block to rotate under the head.
- **Transfer time**, which is the time it takes for moving data to/from the disk block.

Hard disks have a more complicated structure than they used to when these terms were coined, but the estimations provided by these terms remain fairly accurate.

To minimize rotational delay, blocks that are fetched in quick succession should be stored close together. This allows for pre-fetching, where extra blocks of data are fetched in anticipation that they will be read shortly.

2.2 RAID

Redundant Array of Independent Disks (RAID) is a storage system organization that combines multiple physical disks into logical storage units. Common goals of RAID include data redundancy and parallel reads. The features offered by RAID are determined by the RAID level, with level 0 offering no redundancy, level 1 providing data mirroring (i.e. two disks storing same data), and higher levels offering further features such as bit or block level data striping.

2.3 Records and Files

A record is some group of fields storing information about an entity. For example, an employee record may store name, rank, and salary. A file stores a series of records. A record id (rid) is an identifier that can be used to physically locate a record in memory (e.g. page ID and position on that page).

Common structures of files include:

- **Heap File:** These files store records in random order. They are good for write-heavy workloads, but not optimized for searching.
- **Sorted File:** These files sort their records by some key. They are good for range-based and equality-based lookups of records, but updates are slower than for heap files.

2.4 Buffer Management

The buffer manager is a low-level component of a DBMS that moves pages between persistent storage (e.g. disks) and RAM. Data must be stored in RAM in order to be read or written to. DBMS systems typically implement a custom file system, since the default operating system (OS) file system may run into issues with portability, or may have other limitations such as not supporting files spanning multiple disks.

3 Indexing

Files can include indices, which are data structures that speed the retrieval of records searched for by an index's search key fields. For example, if a file stores records of employees and one wants to speed up searching by name, an index can be created with name as the search key. In this example, a name is considered to be a data entry, while the employee is considered to be a data record. Indices can also be created over multiple search keys (known as composite search keys). While indices can speed up reads, they tend to add overhead to writes that slows them down.

3.1 Data Storage in Indices

An index can facilitate retrieval of records by either:

1. Storing a data record next to the data entry. At most one index on a collection of data records can use this technique, since otherwise records are duplicated. Also, large data records may result in fewer data entries per page, which can result in more page fetches.
2. Storing the rid of where the data can be retrieved. This allows more data entries to fit on a page, since (possibly large) data records are not stored next to the data entries. However, a separate page load is necessary to retrieve a data record based on its rid.

3.2 Index Classification

If a search key contains the primary key of a file, it is known as the primary index. Other indices are known as secondary indices.

If the order of data records is the order of data entries of an index, the index is considered clustered. (e.g. see storage example 1 in above section). Clustered indices can greatly speed retrieving records due to a decreased number of page loads. To avoid duplication of records, at most one index in a file can be clustered. However, there can be multiple partially clustered indices, which are indices in which the order of data records is roughly that of the data entries, which can provide similar advantages to clustered indices. For example, if employee salary is a clustered index, employee rank may be partially clustered, since rank is likely correlated to salary.

3.3 Index Implementations

Indices are most commonly implemented using hash tables or tree structures.

3.3.1 Tree-Based Indices

Tree-based indices are implemented using search trees. Since search trees are ordered, tree-based indices support both equality and range searches.

The most widely used search-based indices are B+ trees. These are trees that store alphabetically ordered search keys in internal nodes and data entries in leaf nodes. Internal nodes are typically sized so that each node is the same size as a page in memory, limiting the number of page loads. The number of pointers to child nodes in an internal node is known as a B+ tree's fanout. A large fanout decreases the tree's height.

Tree-based structures may need to adjust their index after every write. Thus, when many writes are done in sequence it can be advantageous to simply sort the entries and then rebuild the B+ tree's index after the writes are complete, decreasing the number of page loads. Such techniques for inserting multiple records efficiently are known as bulk loading.

3.3.2 Hash-Based Indices

Hash-based indices implement search using a hash table with buckets containing data entries. Hash-based indices can support equality searches, but since hash table are unordered, they do not speed up range searches.

Hash-based indices differ in their hashing schemes. Some common hashing schemes are:

- **Static Hashing** uses a fixed number of buckets, and entries point to a chain of data records. This chain can grow long if there are many collisions.
- **Extendible Hashing** is a form of dynamic hashing, which can dynamically grow and shrink the number of buckets to balance memory usage with lowering the risk of collisions. The high-level idea is to split a bucket when it overflows, which may involve doubling the keyspace, and to undo this operation if a bucket becomes empty.
- **Linear Hashing** is another form of dynamic hashing. It avoids the "directory-like" structure of extendable hashing by allowing the number of buckets to increase one at a time instead of growing by doubling the keyspace. It supports overflow entries, but avoids long overflow chains by redistributing entries in a round-robin fashion when a new bucket is added.

3.4 Cost Models

When comparing index and file organization options, it can be helpful to model the cost of common operations (search (equality and range), insert, update, delete) using the following parameters:

- B - number of pages with data
- R - number of data records per page

- D - average time to read or write page

Index and file organization options should be considered with the database's type of workload and data in mind (i.e. workload profiling and data profiling).

3.5 Index-Only Plans

Some queries can be answered without actually retrieving any data records. For example, if a query is simply counting the number of employees with a certain name, then it is sufficient to count the number of data entries in an index using employee name as the search key; no data records need to be accessed, making this an index-only plan.

4 External Sorting

4.1 Motivation

Sorting is a very common problem in computer science. Sample sorting applications in databases include returning sorted data, sorting intermediary results to speed up some other operation such as removing duplicates, and bulk loading B+ trees.

Sorting in databases often deals with very large amounts of data that cannot all fit into RAM. This introduces the need for external sorting algorithms.

4.2 External Merge Sort

Consider a case where N pages of data need to be sorted using B pages of memory (known as buffer memory). The external merge sort algorithm is as follows:

1. Sort the values within in each page in memory. This can be done using any sort algorithm. One variant is to use heapsort to read in B blocks right away to produce sorted runs with an average length of $2B$.
2. Repeatedly use B buffer pages to run merge sort on the merged sublists. Each run produces $\frac{N}{B}$ sorted runs, and each sorted run grows by B pages between sorting rounds.

The total number of sorting rounds (also known as passes) is around $1 + \log_{B-1}((\frac{N}{B}))$. The number of read/write page operations in each pass is $2N$ (i.e. one for loading page into buffer memory, one for writing sorted buffer back to persistent memory).

4.3 Using B+ Trees

Consider a case where a relation to be sorted has a B+ tree index over the sort columns. If the index is clustered, then using the index can be very effective, since the records can be retrieved in order page-by-page. If the index is unclustered, the order of page loading will be seemingly random and result in heavy page thrashing.

5 Evaluation of Relational Operations

5.1 Selections

Selections involve selecting a subset of tuples from a relation based on some condition. Two high-level approaches to selections with multiple conjunct terms are:

1. Find the most selective access path (i.e. the one with the fewest returned tuples), then the most selective access path for the remaining tuples, and so forth.
2. Retrieve the rids for each conjunct term in a condition separately, find the intersection between these rids, and then retrieve the corresponding records.

The simplest physical implementation of a selection is to perform a linear scan over a relation and return those tuples that pass a condition. Indices can be used to speed up selections. Note that unclustered indices can be made to imitate clustered indices by sorting their rids before retrieving records, though this can greatly slow down the speed of retrieving the first few records.

5.2 Projections

Projections remove duplicate rows and unwanted columns from a relation. Removing unwanted columns is quite simple. Removing duplicates can be done by:

1. Sorting, which could be done by modifying the external merge sort algorithm to eliminate duplicates when merging. This implementation has the additional potential benefit that the results are sorted.
2. Hashing, which is done by using a hash function h to partition a result set into subsets small enough to fit in memory. The duplicates can then be removed from each subset.

5.3 Set Operations

Intersection and cross products are a special case of joins, which are covered below. The union and difference set operations can be implemented similarly to projections.

5.4 Aggregate Operations

Aggregate operations (e.g. AVG, MIN, MAX) without grouping typically require scanning an entire relation. If grouped, then a sorting or hashing approach similar to that of the projection operation can be applied so that tuples grouped in the operation can be loaded into memory together.

5.5 Joins

Joining data across tables is a common and potentially very expensive database operation. Various join algorithms are available and their performance can vary greatly depending on the joined data, the available indices, and the join conditions.

The below examples consist of joining tables R and S over some condition. For cost analysis, M is the number of pages in R, N is the number of pages in N, and p_R is the number of tuples per page in R.

5.5.1 Simple Nested Loop Join

The simple nested loop join tables R and S is as follows:

1. Loop through each tuple r in R.
2. For each r, loop through each tuple s in S.
3. If the join condition evaluates to true for r, s, then we add (r, s) to the output.

R is considered to the be outer relation, and S is considered to be the inner relation. When done naively by just looping through one record at a time, the number of IO operations is around $M + p_R MN$. The p_R term can be removed by finding matches between R and S one page at a time rather than one record at a time, making the number of IO operations $M + MN$. This variant is known as page-oriented or block nested loop join.

5.5.2 Index Nested Loop Join

Index nested loop joins differ from simple nested loop joins in that they can use an index of the inner relation. The number of IO operations is $M+(Mp_R)$ cost of findings matching tuples in S.

5.6 Sort Merge Join

Sort merge join involves sorting R and S on the join column. R and S are then scanned sequentially to look for matches. The cost of sorting R is $M \log M$ and the cost of sorting S is $N \log N$. The cost of scanning after sorting is usually around $M + N$, though could be as high as MN.

Sort merge join can be altered so that the joining occurs during the merging portion of external merge join.

5.6.1 Hash Join

In a hash join, a hash function h is used to partition relations R and S. These partitions should be small enough to fit in main memory, and they are then loaded in pairs, after which

matches are found between them. To further speed lookup, a hash table can be built for each partition of R, though this requires a bit more memory.

Hash join has a cost of $2(M + N)$ during the partitioning phase, and $M + N$ during the matching phase. This is similar to the cost of sort merge join. Hash join is highly parallelizable and performs better than sort merge join if relation sizes vary greatly. However, sort merge join is less susceptible to data skew.

5.6.2 General Join Conditions

All of these algorithms work for equality-based join conditions, including multiple conjunct equality terms. Sort merge join and hash join do not work for inequality conditions (e.g. greater than). If using an index nested loop join for such inequality conditions, it should be clustered. If there is no clustered index, then the block nested loop join is likely the best option.