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 Indexing

2.1 Files and Indices

A record id (rid) is an identifier that can be used to physically locate a record in memory. A file stores a series of records. 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.

As an alternative to indices, some non-indexed structures of files include:

- **Heap File:** These files store records in random order. They are good for write-heavy workloads, but bad for reading due to their lack of indices.
- **Sorted File:** Good for range-based and equality-based lookups of records, but updates are slower than for heap files.

2.2 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.

2.3 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.

2.4 Index Implementations

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

2.4.1 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.

2.4.2 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.

2.5 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 in mind.

2.6 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.