CMU 15-445 - Database Systems

CMU 15-445/645 - Database Systems

Taught by Prof. Andy Pavlo in Fall, 2019 and Fall, 2023.
The first 7 lectures are from the course taught in Fall 2019, the rest are from the course taught in Fall 2023.

Course Description

This course is on the design and implementation of database management systems. Topics include data models (relational, document, key/value), storage models (n-ary, decomposition), query languages (SQL, stored procedures), storage architectures (heaps, log-structured), indexing (order preserving trees, hash tables), transaction processing (ACID, concurrency control), recovery (logging, checkpoints), query processing (joins, sorting, aggregation, optimization), and parallel architectures (multi-core, distributed). Case studies on open-source and commercial database systems are used to illustrate these techniques and trade-offs. The course is appropriate for students with lit systems programming skills.

Course Website (2019) Course Website (2023) Lecture Videos (YouTube)

Lectures

1. Introduction & Relational Model
2. Advanced SQL
3. Database Storage 1
4. Database Storage 2
5. Buffer Pools
6. Hash Tables
7. Tree Indexes

Introduction & Relational Model

DBMSs

A database is an organized collection of related data that models some aspect of the real world (e.g., modeling the students in a class, or a digital music store). A DBMS is a software that allows applications to store and analyze the information in a database. It allows the definition, creation, querying, updating, and administration of databases.

DBMSs have 2 layers - the logical layer and the physical layer. The logical layer defines which entities and attributes the data stored in the database will have, and the physical layer is how those entities are actually stored (like the particular data structure used to store the data).

Before relational databases, the physical layer was defined in the application code itself, which made the database much harder to manage. If the structure of the data or the requirements of the application changed, the whole physical layer needed to be rewritten.

Relational Model

Ted Codd proposed the relational model in 1970. The relational model has 3 key points -

1. Store database in simple data structures.
2. Access data through a high-level query language.
3. The physical storage of data is left up to the implementation.

The relational model is an example of a *data model*. A data model is a collection of concepts for describing the data stored in a database. A *schema* is a description of a particular collection of data using a particular data model.

The relational model defines three concepts - 1. Structure - Data is distributed into tables and tables can be related to each other. The structure also describes what kind or type of data each table can hold. 2. Integrity - The database's contents always satisfy some constraints, which can either be defined by the designer or the database itself. 3. Manipulation - Describes how the contents of the database can be modified.

Relational Algebra

Relational algebra gives a set of operations we can perform on relations to produce new relations. You can think of a relation as a table in a database. Relational algebra operators take a relation as an input and produce another relation.

The basic few relational algebra operators are -

1. Select ($\sigma_{\text{predicate}}$(R)) - Acts like a filter. Selects tuples from the relation that satisfy the predicate.
2. Projection ($\pi_{\text{A1,A2,...}}$(R)) - Returns the same relation but only with the attributes A1, A2, ... selected from the relation.
3. Union (R $\cup$ S) - Produces a new relation that contains all the tuples in at least one of the relations (the relations need to have the same attributes).
4. Intersection (R $\cap$ S) - Produces a new relation that contains only those tuples that exist in both R and S (the relations need to have the same attributes).
5. Difference (R - S) - Produces a new relation that contains all tuples that appear in the first relation but not in the second relation (the relations need to have the same attributes).
6. Product (R $\times$ S) - Product takes in two relations and outputs a relation that contains all possible combinations for tuples from the input relations.
7. Join (R $\bowtie$ S) - outputs a relation that contains all the tuples that are a combination of two tuples where for each attribute that the two relations share, the values for that attribute of both tuples is the same.

Advanced SQL

Aggregates

SQL has a few aggregate functions that take a bag of tuples as the input and produce a single scalar value as the output. These can only be used on the left-hand side of a query, in the output list of the `SELECT` clause.

Here is the example database we will be using for this part of the lecture -

```
CREATE TABLE student (
    sid INT PRIMARY KEY,
    name VARCHAR(16),
    login VARCHAR(32) unique,
    age SMALLINT,
    gpa FLOAT
);

CREATE TABLE course (
    cid VARCHAR(32) PRIMARY KEY,
    name VARCHAR(32) NOT NULL
);

CREATE TABLE enrolled (
    sid INT REFERENCES student (sid),
    cid VARCHAR(32) REFERENCES course (cid),
    grade CHAR(1)
);
```

For example, we have this query which gets the number of students in the students table with a '@cs' login -

```
SELECT COUNT(*) FROM student WHERE login LIKE '%@cs';
/* or */
SELECT COUNT(login) FROM student WHERE login LIKE '%@cs';
/* or */
SELECT COUNT(1) FROM student WHERE login LIKE '%@cs';
```

We can use multiple aggregate functions in a single query -

```
SELECT AVG(gpa), COUNT(sid) FROM student WHERE login LIKE '%@cs';
```

If you use an aggregate function in your query and you also select some other columns, you will need to `GROUP BY` those columns. For example, if you `AVG(student.GPA)` and `student.id` in the same query, you will need to `GROUP BY student.id`. This is because you want to get one row for each student, containing that student's average

GPA, which means you want to group each student into a row of their own, look at all of their GPA values, and take the average.

Another example - ~~sql SELECT AVG(s.gpa), e.cid FROM enrolled AS e, student AS s WHERE e.sid = s.sid GROUP BY e.cid;~~

Database Storage 1

Memory Hierarchy

We will assume the database management system will have to deal with 2 levels of memory - volatile and non-volatile. This lecture and the next few lectures are focussed on how the data is actually stored in the file system.

File Storage

We want to give the user the impression that the entire database is stored in the main memory, and not stall when the user requests data from outside the main memory, even though calls to disk are slow.

We want to implement something like a virtual memory.

Now, the operating system can already do this well, but we almost never want to leave it up to the OS to handle this for us, because we can manage memory much better since we have extra information about the queries that are being run and that can be run, and we can use that information to perform spatial and temporal optimizations. Moreover, some queries may require specialized memory management for correctness.

A DBMS stores a database in memory as a file (sqlite) or multiple files. The OS does not know anything about what is inside those files, only the DBMS can interpret them correctly.

The DBMS has a *storage manager* that is responsible for managing these files. It stores the database in these files in blocks of data called pages. A page is just a block of data that can contain any kind of information. There are three kinds of pages -

1. Hardware pages - The blocks exposed by the SSD/disk itself on the hardware level. Usually 4kB.
2. OS page - The blocks as defined by the OS (4kB).
3. Database page - The blocks as defined by the DBMS (0.5 to 16kB).

Heap File Organization

There are a few ways to organize pages in memory, and the one covered in the lecture is the heap file organization. A heap file just means that pages are stored in memory in an unordered way, and tuples are stored inside those pages.

We maintain something called a page directory, which is basically a hash table that maps page ids to the location of the pages in memory. It can also contain metadata about the page.

Page Structure

The data contained in the page itself can be stored in two ways - slotted pages and log-structured pages.

Slotted pages are pages which contain "slots" for tuples to go into. The page has a header that stores metadata about the page, and the top of the page contains a "slots" array that maps slot locations to an offset in the page. The tuples are stored starting from the bottom of the page working up. The page is said to be full when the slots array and the tuples touch.

Log structured pages store only logs, i.e, records of how the page was modified. The data present in the page can then be inferred from the logs during a read. This results in fast writes, but could lead to slow reads. However, there are a few techniques to make reads practical as well.

Database Storage 2

Data Representation

This section describes how the DBMS stores the values inside the tuples in memory. Tuples' data is essentially just byte arrays. It is up to the DBMS to know how to interpret those bytes to derive the values for attributes. A data representation scheme is how a DBMS stores the bytes for a value. There are four main types that can be stored in tuples: integers, variable precision numbers, fixed point precision numbers, variable length values, and dates/times.

Integers, Floats, Reals

These are usually just stored in the same way C/C++ store them, as specified by the IEEE-754 standard.

Fixed Point Precision Numbers

The DBMS defines how it will handle fixed point precision numbers, and implements an API to access them itself. It does not rely on CPU instructions, since they have rounding errors. Instead, the entire precision of the number is stored in the representation, along with additional metadata telling the DBMS how to interpret the value.

These types are used when rounding errors are unacceptable, for example in scientific or financial applications. However, handling these types is much slower than floating point numbers.

Variable Length Data

These are data types that consist of a varying amount of data that needs to be stored. For example, `TEXT`, `VARCHAR`, `BLOB`, `VARBINARY`. Has a header that keeps track of the length of the string to make it easy to jump to the next value. Most DBMSs do not allow a tuple to exceed the size of a single page, so they solve this issue by writing the value on an overflow page and have the tuple contain a reference to that page.

Some systems will let you store these large values in an external file, and then the tuple will contain a pointer to that file. For example, if our database is storing photo information, we can store the photos in the external files rather than having them take up large amounts of space in the DBMS. One downside of this is that the DBMS cannot manipulate the contents of this file.

Datetime Data

These are usually just represented as an integer equal to the number of milli or micro seconds passed since the UNIX epoch.

Workload Types

Depending on the type of work you want to do with your data, systems can be divided into two types - Online Transaction Processing (OLTP), and Online Analytical Processing (OLAP).

OLTP

OLTP basically means you want transactions on your database to be fast. It is usually used when your application is read heavy, and you want to collect potentially large amounts of data from your frontend efficiently.

Characteristics of OLTP systems are - 1. Fast, short running operations. 2. Queries usually only operate on a single entity at a time (hence joins are rare). 3. Application is read-heavy. 4. Usually collect more data than they need to analyze.

OLAP

OLAP means that you want joins and other complex operations involving multiple parts of your database to be fast, usually because you want to perform analytics.

Characteristics of OLAP systems are - 1. Long running, complex queries accessing multiple columns, tuples, and tables. 2. Long scans through entire tables. 3. Analytical and exploratory queries. 4. Usually analyze and read more data than they write.

Storage Models/Methods

The relational model does not specify that you should store your tuples contiguously. There are two types of storage models - Decomposition storage model (column storage) and N-Ary storage model (row storage).

NSM

NSM stores all the columns of a single tuple contiguously in the same page. This approach is ideal for OLTP workloads where transactions tend to operate only an individual entity and insert heavy workloads. It is ideal because it takes only one fetch to be able to get all of the attributes for a single tuple.

Its advantage is fast inserts and updates, and it is good for queries that need the entire tuple. Its disadvantage is queries that use only a small portion/number of columns of the tuple.

Therefore, row stores are usually not used for OLAP systems.

DSM

DSM stores a single attribute or column for all tuples contiguously in a single page.

Locks vs. Latches

Locks and latches mean different things in a DBMS than in an OS.

Locks are - 1. Used to protect the logical contents of the database from other transactions. 2. Held for transaction duration 3. Help to roll back changes

Latches are - 1. Used to protect internal data structures from other threads 2. Held only for operation duration 3. Do not need to be able to roll back changes

Buffer Pool

A buffer pool is just an area in memory that the DBMS reserves to read pages from disk. The buffer pool consists of discrete "slots" of memory we call frames that are the same size as a page. The DBMS reads pages from disk and brings them into frames.

A buffer pool maintains the following metadata - 1. Page table - A hash table mapping page ids to memory location in the buffer pool 2. Dirty flag - A flag indicating whether a page has been modified. 3. Pin counter - The number of threads currently accessing that page. This counter prevents pages that are being accessed from being evicted.

Now that we have introduced buffer pools, we can make the following optimizations - - Multiple buffer pools - We can have multiple buffer pools for different purposes. Each buffer pool can have its own replacement policy and some other properties that may help. For example, having a dedicated buffer pool for each table can help with some particular queries. - Pre-fetching - The DBMS can pre-fetch pages into the buffer pool if it decides that it needs to based on the query plan. The OS can also do this for simple queries, but not in all situations (e.g, with indexes on a query that operates on a certain range of pages). - Scan sharing - If a query is performing a scan over a table, and another query comes in from another thread that also needs to perform the same scan, it can join the scan that the first query is performing from the page that the scan has reached now.

Replacement Policies

Once we fill our buffer pool, which can happen pretty quickly for large queries, we will need to evict an existing page from the pool to make space for a new page. The way we decide which page to evict is called the replacement policy. This is an extremely important component of the DBMS, because it can make a huge difference in performance. Commercial database systems have a very complex replacement policy as compared to open source systems.

There are a few common replacement policies, but the most popularly used ones are LRU and clock. You already know what LRU is so I won't write about it.

Clock is an approximation of LRU that doesn't need you to store the last accessed time stamps. You maintain a circular queue with a pointer pointing to the "current" page. Each page has a single bit of metadata that indicates whether that page has been accessed since the last time the pointer was here. The pointer always moves forward (or "clockwise"), and when it wants to evict a page, it checks the reference bit for that page. If that bit is 1, that means that the page was accessed fairly recently, and we don't evict that page, but we set its bit to 0. We evict the first page we find that has its reference bit set to 0.

B-Trees and Indexes

A table index is a replica of a subset of a table's columns that is stored in such a way that it allows the DBMS to find tuples faster than sequential scans.

Indexes dramatically speed up some types of queries, but they also take up space, and need to be kept in sync with the data. Too many indexes can make reads fast but writes slower.

The query optimizer decided which index to use when it receives a query.

B+ Tree

A B+ Tree is a self-balancing tree data structure that keeps data sorted and gives $O(logn)$ search, sequential access, insertion, and deletion.

A B+ Tree is used in almost all DBMSs which support order-preserving indexes.

A B+ Tree comes from a class of data structures called B Trees. It has the following properties - 1. It is perfectly balanced (every leaf is at the same depth). 2. Every inner node other than the root is always at least half full. 3. Every inner node with k keys has k+1 children.

That is, if we have a B+ Tree of degree M, that means its nodes can contain at most $M-1$ keys and M children. Property 2 says that every inner node has between $M/2-1$ to $M-1$ keys.

Andy then explains the insertion and deletion algorithms in a B+ Tree, which I won't describe here. But they are pretty much what you would expect.