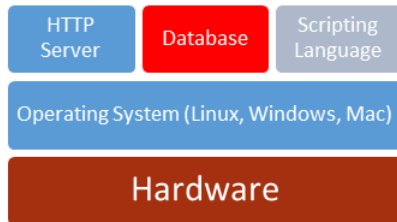


4.1: Database Types

While there are a number of databases available for use like MySQL, node.js, and Access, there is an additional list of the types of database structures each of these belong to. These types each represent a different organizational method of storing the information and denoting how elements within the data relate to each other. We will look at the three you are most likely to come across in the web development world, but this is still not an exhaustive representation of every approach available.



Flat File

Flat files are flat in that the entire database is stored in one file, usually separating data by one or more delimiters, or separating markers, that identify where elements and records start and end. If you have ever used Excel or another spreadsheet program, then you have already interacted with a flat file database. This structure is useful for smaller amounts of data, such as spreadsheets and personal collections of movies. Typically these are comma separated value files, .csv being a common extension. This refers to the fact that values in the file are separated by commas to identify where they start and end, with records marked by terminating characters like a new line, or by a different symbol like a colon or semi-colon.

The nature of all data being in one file makes the database easily portable, and somewhat human readable. However, information that would be repeated in the data would be written out fully in each record. Following our movie example, this could be the producer, studio, or actors. They have what we call a one-to-many relationship with other data in the database that cannot be tracked in this format.

Drawbacks to this format can affect your system in several ways. First, in our example, we would enter the studio name into each record for a movie made by that studio. If the person typing miss-typed an entry, it may not be found when users search the database, skewing search results through missing information. This often results in users creating new entries for a record that appears not to exist, causing duplication. Beyond search issues, every repetition is more space the database requires, especially when the value repeated is large. This was more of an issue when data was growing faster than storage capacity. Now, with exception to big data storage systems, the average user's storage space on an entry level PC typically surpasses the user's needs unless they are avid music or movie collectors. It is still important to consider when you are dealing with limited resources, such as mobile applications that are stored on smart phones that have memory limitations lower than that of desktops and servers.

Another issue with these files is the computational effort required to search the file, edit records, and insert new ones that are placed somewhere within the body of data as opposed to the start or end of the file.

Finally, flat files are not well suited for multiple, concurrent use by more than one party. Since all of the data is in one file, you are faced with two methods of interaction. The first approach is allowing anyone to access the file at the same time, usually by creating a local temporary copy on their system. While this allows multiple people the ability to use the file, if more than one party is allowed to make changes, we risk losing data. Say User 1 creates two new records while User 2 is editing one. If User 2 finished first and saves their changes, they are written back to the server. Then, User 1 sends their changes back, but their system is unaware of the changes made by User 2. When their changes are saved, User 2's changes are lost as User 1 overwrites the file. This can be prevented by checking last modified timestamps before allowing data to be written, but the user "refreshing" may have conflicts between their edits and the edits from another user, when the same record is changed.

The alternate approach to allowing multiple users is to block multiple users from making changes by only allowing one of them to have the file open at a time. This is done by creating a file lock, a marker on the file the operating system would see, that would block other users from using an open file. This approach does not support concurrent access to the data, and again even allowing read rights to other users would not show them changes in progress that another user has not completed and submitted. Another downside to this approach is what is called a race condition—where multiple systems are trying to access a file, but are unable to do so because another has the file locked, stalling all of the programs trying to access the data.

This was a key element in a large scale blackout of 2003 that took place in the Northeast United States and part of Canada. A summer heat wave created significant strain on the power system as demand for air conditioning increased, resulting in the emergency shutdown of a power station. This station happened to be editing its health status in a shared file between itself and other stations, a method used to allow other stations to adjust their operational levels in response to their neighbors. The purpose of this file was to act as a protection method, warning of potential spikes or drops in power at individual facilities. When the plant using the file shutdown, the file remained locked as the computer using it did not have time to send a close file command. Unable to properly close the file with the systems down, other stations were unaware of the problem until power demand at their facilities rapidly increased. As these stations could not access the file due to the lock, a warning could not be passed along. Since the power stations were under increasing strain with each failure, a cascading affect occurred throughout the entire system. Admittedly an extreme result of file lock failure, it is a very real world example of the results of using the wrong tools when designing a system.

Structured Query/Relational Database

Structured query databases can be viewed similar to flat files in that the presentation of a section of data can be viewed as its own table, similar to a single spreadsheet. The difference is that instead of one large file, the data is broken up based on user needs and by grouping related data together into different tables. You could picture this as a multi-page spreadsheet, with each page containing different information. For example, continuing with our movie example, one table would contain everything about the studio—name, opening date, tax code, and so on. The next table would contain everything about the movies—name, release date, description, production cost etc. Finally we might have a table for actors, producers, and everyone else involved. This table would have their information like birthday, hometown, and more.

What we do not have yet is a way to link these elements together. There is also a lot of information we *do not* want to include, because we can determine it from something else. For example, we do not want to store the actors age, or we would have to update the table every year on their birthday. Since we already have their birth date, we can have the server do the math based on the current date and their birth date to determine how old they are each time it is asked.

To address relating an actor in our people table to a movie they were in from the movie table, as well as to the studio that made the movie in the studio table, we use a structured query. Structured query is a human readable (relatively) sentence style language that uses a fixed vocabulary to describe what data we want and how to manipulate it. Part of this comes from adding extra tables. Since one actor can be in many movies, and each movie can have many actors, we have a many-to-many relationship between them. Due to this, we create an extra table where each row represents a record of an actor and a movie they were in. Instead of putting their full names into this table, we put the row number that identifies their information from their respective tables. This gives us a long, skinny table that is all numbers, called an “all-reference table,” as it refers to other tables and does not contain any new information of its own. We will see this in action soon.

We can use our query language to ask the database to find all records in this skinny table where the movie ID matches the movie ID in the movie table, and also where the movie name is “Die Hard.” The query will come back with a list of rows from our skinny table that have that that value in the movie ID column. We can also match the actor IDs from a table that pairs actors with movies to records in the actor table in order to get their names. We could do this as two different steps or in one larger query. In using the query, we recreate what would have been one very long record in our flat file. The difference is we have done it with a smaller footprint, reduced mistyping errors, and only see exactly what we need to. We can also “lock” data in a query database at the record level, or a particular row in a database, when editing data, allowing other users access to the rest of the database.

While this style can be very fast and efficient in terms of storage size, interacting with the data through queries can be difficult as both one-to-many and many-to-many relationships are best represented through intermediary tables like we described above (one-to-one relationships are typically found within the same table, or as a value in one table directly referencing another table). In order to piece our records together, we need an understanding of the relationships between the data.

MySQL

Structured query language databases are a very popular data storage method in web development. While different approaches are emerging to address big data issues, the concepts you learn by studying structured query can help you organize data in any system.

MySQL, commonly pronounced as “my seequel” or “my s q l,” is a relational database structure that is an open source implementation of the structured query language. The relational element arises from the file structure, which in this case refers to the fact that data is separated into multiple files based on how the elements of the data relate to one another in order to create a more efficient storage pattern that takes up less space.

In a traditional LAMP, MySQL plays the role of our data server, where we will store information that we want to be able to manipulate based on user interaction. Contents are records, like all the items available in Amazon's store. User searches and filters affect how much of the data is sent from the database to the web server, allowing the page to change at the user's request.

History

MySQL began when developers Monty Widenius and David Axmark attempted to use the mSQL database system to interact with their own tables they had created using low-level routines in ISAM. Their testing did not reveal the speeds or flexibility they wanted, so they created their own similar API, and dubbed it MySQL after co-founder Monty's daughter My.

After an internal release in 1995, MySQL was opened to the public with minor version updates spanning 3.19 to 3.23 from 1996 to 2000. Their next major version release, 4.0, arrived as beta in 2002 and reached production in 2003. Their next major release arrived as 5.0 in 2005 and included the addition of cursors, stored procedures, triggers, and views. These were all significant additions to the toolset.

In 2008, Sun Microsystems acquired what was then called MySQL AB with an agreement to continue development and release of the software as a free and open source item. When Sun was acquired by Oracle in 2010, MySQL was part of the package deal, under the same requirements. There have been some arguments over whether or not the spirit of the agreement between MySQL and Sun has been fully upheld by Oracle, including complaints from Widenius himself.

Structure

We organize data in MySQL by breaking it into different groups, called tables. Within these tables are rows and columns, in which each row is a record of data and each column identifies the nature of the information in that position. The intersection of a row and column is a cell, or one piece of information. Databases are collections of tables that represent a system. You can imagine a database server like a file cabinet. Each drawer represent a database in our server. Inside those drawers are folders that hold files. The folders are like our tables, each of which holds multiple records. In a file cabinet, our folders hold pieces of paper or records, just like the individual rows in a table. While this may seem confusing now, we will see it in action soon; this is the approach we will focus on for this section of the text.

NoSQL

NoSQL databases represent systems that maintain collections of information that do not specify relationships within or between each other. In reality, a more appropriate name would be NoRel or NoRelation as the focus is on allowing data to be more free form.

Most NoSQL system follow a key-value pairing system where each element of data is identified by a label. These labels are used as consistently as possible to establish common points of reference from file to file, but may not be present in each record. Records in these systems can be represented by individual files. In MongoDB, the file structure is a single XML formatted record in each file, or it can be ALL records as a single XML file. Searching for matches in a situation like this involves analyzing each record, or the whole file, for certain values.

These systems excel when high numbers of static records need to be stored. The more frequently data needs to be changed, the more you may find performance loss here. However, searching these static records can be significantly faster than relational systems, especially when the relational system is not properly normalized. This is actually an older approach to data storage that has been resurrected by modern technology's ability to capitalize on its benefits, and there are dozens of solutions vying for market dominance in this sector. Unless you are constructing a system with big data considerations or high volumes of static records, relational systems are still the better starting place for most systems.

Learn more

Keywords, search terms: Database types, data structures, data storage

Node.js: <http://nodejs.org/>

The Acid Model: <http://databases.about.com/od/specificproducts/a/acid.htm>

Key-Value Stores, Marc Seeger: http://blog.marc-seeger.de/assets/papers/Ultra_Large_Sites_SS09-Seeger_Key_Value_Stores.pdf

4.1: Database Types is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

- **4.1: Database Types** by Michael Mendez has no license indicated.