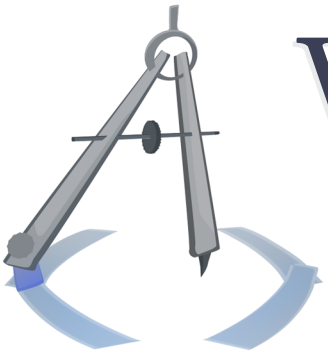


Website Architecture

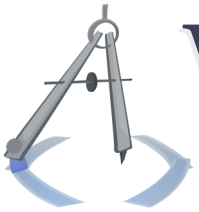
Lezione 12

Relational Databases

A conceptual explanation of databases and then relational databases, including their intended purpose and their basic features.

Michael Serritella

Summer 2010



Intro to databases

Databases are basically persistent places to keep data, and database management systems are programs which help to retrieve and modify the data as quickly, easily, and accurately as possible. The data could be thought of in small or large pieces - boolean flags, integers, or large objects. The problem of storing things and looking them up later is not unique to databases, and we will see a sort of extended metaphor that uses a filing cabinet as an example.

Desired features

When using a database, espeically in a Web environment, we have certain needs and expectations:

Persistence The data persists longer than a page view and longer than a session - as long as needed.

Simplicity of access We can access the data by simply declaring that we want to use it, in some kind of well-formed way. We don't want to have to hack up our own text file and some arbitrary format to store data.

Concurrency We want to be able to service infinitely many people at once, even if some of them are reading from and writing to the same data at (almost) the same time.

Integrity and authentication We want to lock the whole database with a username & password, and we want to create additional roles within the database that have exact privileges, from specific to general tasks.

We can't get all of this from hacking up our own text file. So let's use a database.

Basic example

In this section, we will see an analogy which illustrates all the basic use of a database, including issues related to finding things quickly.

Intro to the file cabinet analogy

Databases are usually explained as tables of data, with rows and columns, but there are far more natural analogies. We will use the analogy of a filing cabinet. Let's say you have a **filing cabinet** with **three drawers**. In each drawer is a row of files; let's say they are drawers for **Customers**, **Orders**, and **Inventory**. Each file in the drawer has **a bunch of data fields** (atomic pieces of data) that pertain to a particular customer, particular order, etc.

In this analogy, the file cabinet is the database, and perhaps a secretary or office worker is a database management system, since he/she will actively get and set the data that you want, operating upon the file cabinet.

Basic selection

Suppose you want a file of a particular customer. So you tell your assistant to go get the file! Simple. Pretty much. Without any particular organization scheme, this is clearly possible, though it is slow; we will see ways to improve this.

Selecting specific fields

More often than not, you don't want the entire file. You want a specific data field from the file. What was the customer's age? City of residence? etc. Maybe you want more than one field. You can specify that, too. Naturally, your assistant first has to find the file, and then picking out the relevant parts is easy.

Selecting a set of records

Maybe you want more than one file. You want all customers named Johnson, or you want all male customers. Then, you really only care about their name, age, and phone number. Databases can also accommodate this kind of request.

Keys as unique labels

So far, we have a plain filing cabinet, with no particular means of organization. The files don't even have little tabs at the top. But the tabs are a good idea, right? Let's do that.

Before, your assistant had to look inside of each file to see if it was the one you wanted - to see if the name or gender matched, for instance. Now, let's add a small label that uniquely identifies the file. In database terms, this is a **key**, roughly speaking.

Let's use the customer's last name and first initial as the key (e.g. "Wilson, W."). Not too bad.

Once we have a unique identifier for the customer file, we can refer to the customer file by its key instead of trying to refer to it by a more exhaustive, descriptive name ("The Wilson file, guy named Woodrow, .."). So the key becomes quite useful when describing the files in shorthand notation.

Real-world problems

Well.. this is a pretty good idea for a chump family business. Maybe you don't ever have two people with conflicting names, and maybe there are no issues of privacy if these names are thrown around the office.

What if you have a million customers? Ten million or more? You can't have thousands of files in your file cabinet labeled "Smith, J.". You need a better scheme.

Similarly, what if this key is visible to people who don't need to know it? Couriers; other customers; your chump nephew who works there? If the file is laying around with its tab visible, it exposes potentially sensitive information.

So, what's a solution? We need a sterile, meaningless label that will not leak too much information if it is passed around by too many people. We need.. customer ID numbers. Preferably random or otherwise meaningless[1].

[1] In practice, these are almost always sequential numbers, though random ones are preferred from a security perspective.

In computer science, with real databases, this is the only way to go[2]. You may think that you can use some combination of last name, first name, etc., to get a viably unique key, but it wouldn't be private, and it may break part of the system if you were to change some of the data that comprises the key. Use keys that have no other meaning or purpose - use unique numbers. Hence, the key is sometimes called **Uno** ("unique number"? origin unknown) or simply **ID**.

Indices

So far, we have a file cabinet with files that have labels, but the files are still in no particular order. Obviously, the first thing that a human would do is sort them. Usually alphabetized. We will see a generalization of this process that allows for different types of sorting than alphabetical.

Alphabetized

But first - why not. Let's sort them alphabetically. This is actually done in databases, after all.

[2]Not really; these types of keys are "surrogate keys." The decision to use them is a design choice and is technically still open for debate.

You may notice that sorting things in real life isn't free. It takes time and effort. Sorting things alphabetically - or in some natural order, like by date - is one of the slower ways to sort things. We have a way to quantify this: if there are n files, it takes about $(n\ log_2 n)$ operations to do this kind of sort.

After the files are sorted alphabetically, how do you look for them? The same way you look for words in a dictionary: Start somewhere; go left; go right; go left again; go right again.. etc. Narrow it down until you find the exact item. Finding the item doesn't take too long.

In computer science, we can't just start somewhere; we have to be specific. We start in the exact middle, and each time, we go left or right for 1/2 the remaining distance. So, if there are n items, it takes about $log_2 n$ operations to find the item. For a thousand items, that's only 10 lookups, and for 4 billion, that's only 32. This kind of search is called a **binary search**, and in real databases, when we organize the files in such a way that they can be searched with binary search, we usually use what is called a **binary search tree**.

Hashing: Organization by an arbitrary function

There is actually a sometimes-better way of organizing files that is a little removed from the example of the filing cabinet. We can actually use hash functions for quicker lookup times.

With hash functions, an input is mapped to a seemingly random number (though the number is consistent between multiple function calls). Let's see how we can exploit that for data organization, using a toy example of a hash function.

Let the function be:
`The number of consonants in a string`.

Let's apply this to the customer's last name. Conceivably, this number is unlimited, since we could have very long names. But let's bound it, which is what we actually do in practice. Let's say that we take this value and apply `modulo 4`. Then, all of our inputs will have approximately randomly distributed values in [0,3].

Now, let's organize our file-cabinet files into four segments:

The front quarter of the drawer, the second, the third, and the back quarter of the drawer (perhaps we have simple cardboard dividers).

Now, let's look up customer Philbin. This has a value `5`, whose residue mod 4 is `1`. So, we will look in the "1th" quadrant (the second one). Now, we have to manually look through that quadrant. But we have saved 3/4 of the lookup time.

How about an improvement? Well, our actual hash value is infinitely long, and we have to shorten it. What if we shorten it to $n/2$, half of the number of items we have? Then we will effectively have a qudarant, sort of (a "bucket"), for each pair of values. We know which one to go to immediately (e.g. "bucket #33"), and then from there, we search the items of the bucket, of which there are only two. Win! Now, what if we made n buckets - then the lookup is still immediate, and we only have to check one item once we get inside the bucket. Infinite win?

Sort of. Almost, really. In reality, (general-purpose) hash functions are not perfect, and more than one item may end up in a bucket. And there are other practical drawbacks that

we'll see, but hashes are generally considered to be very fast, great organization tools when they are applicable.

Applicability of indices

You've seen some types of indices now. You should know that they won't always save the day. They are only applicable in some situtations. If you're searching for customers over 55 years of age, while your only index is on the customer's last name.. Nope! Naturally, you have to have an index on a *search* term if you want to take advantage of it.

A comparison

So, how do these stack up? Should you always use hashing? Well, in reality, no. There's one area in particular that binary search does better.

Imagine a hash function that gives nearly random output for each input (though it is consistent, of course). "John", "Johnson", and "Johnston" would not be anywhere near each other. In real life, you often want to grab a group of files with related

criteria, such as "All customers starting with 'J'" or "All customers starting with 'John'". If you're using a binary search, you only have to find the first one in the sequence, and then you just grab a stack of files. Using a hash, you have to hash each one, and that can add up.

In real life, different types of disks actually exhibit characteristics that favor one or the other. Hard disks favor binary search, and solid-state disks and flash memory favor hashing. We will see more on this later.

Multiple indices

It is also possible to have two indices at once! You can't exactly sort the data in two orders at once, but you can do a next-best thing. This also steps outside of the usual organization of a filing cabinet, but you could do it.

Say that you want to combine the two indices. Let's say you want an index on the customer's last name and you also want an index on the customer's age. Maybe the first is a binary search and the second is a hash.

You can sort the data by the last name to take care of the binary search. But what about the hash? Well, you can create buckets, just like usual, and you can hash the actual data (the age) and see what bucket it goes into. But.. in the buckets, you have *pointers* to the data. You maybe have an index card that says which file, in order ("#59"), is the requested file.

This is obviously indirect and a little less efficient than if the hash were the only index and the actual files were in the buckets.

In real databases (that are well-designed), you can have as many simultaneous indices as you want.

Too many indices?

But how much is too much? Can you ever have too many indices?

Yes. Think about the filing cabinet. What happens when you insert a new file in real life? You have to find its place to the

primary index. And you have to label its position w/r.t. all of the other indices (like making index-card pointers). What if your buckets get too large? Might you have to manage the hash index somehow?

You *can* have too many indices, since each one makes the drawer (table) harder to manage upon an insert or update.

Tables, rows, and columns

In the standard description of databases, we don't use filing cabinets for some reason. We use **tables**, **rows**, and **columns**. To reiterate: The file cabinet is the database, the drawers are tables, the files are rows, and their data fields are columns. Each of these things has a name except for the row. An example table name might be something simple like **Customers**, and a column name would be **FirstName**. The data is usually presented like a spreadsheet.

If you want, you could conceive of the rows as objects in object-oriented programming, where the columns are their variables (or "fields" or "properties"). There is no special reason why the table analogy is most appropriate.

Advanced: The relational model

In real life, we usually conceive of different, isolated types of files which may have links to each other. It's the same old story of object-oriented programming - just a variation on a theme. So we have devised databases which intelligently manage different tables (drawers) which have some kind of relationship, including the facilitation of lookups that involve both tables.

Example of related data

Say that we want to use one of our other drawers - the **orders** drawer. Customers make orders, so they are clearly related in some way. In a given order, we may want to store the customer's name. It seems simple, but is there a smart way of doing this?

A naïve solution

Naively, you would do what almost every business in the world does, including some primitive database systems. In your order

file, you would simply write the customer's name. This is just a duplicate of the customer's name in the customer file.

The relational solution

Now we shall see the relational model, which provides a mostly-more-elegant solution to the problem. In the order file, *store the key* of the customer (e.g. CustomerID), so that the customer's file can be retrieved and the customer's name retrieved from there.

There are some strong advantages and disadvantages to such a scheme. Can you name some of them?

Joins: Lookups across multiple tables

You often want a sort of generalized lookup that spans across multiple tables. For instance, "Give me the orders placed in June 2008, with all the customers' names and phone numbers." How would you execute this lookup in real life?

Well, you may do it any number of ways, though databases

typically do them pretty straightforwardly[3]. Start with the orders table. Go through the orders and find any that occurred in June 2008, perhaps using an index if there is one applicable. For each file you find, keep it in your left hand, and then go through the customers table with your right hand. When you find the right customer with your right hand, group the two files together (maybe stack them) and add them to the output pile.

Multi-joins

It's quite possible that you want to pull from more than two tables. No problem. Same process - just imagine you have more hands. Iterate through each table, pick out an applicable file, hold on to it, move on to the next one, hold on to it, etc.

[3]At least conceptually.

Languages, DBMSes, and APIs

Now for an introduction to the next step. What are database systems on computers, and how do we usually access them, especially from websites?

SQL

SQL is the language which is used to describe these lookups. You describe which rows you want, which columns of them you want to see, from which tables, and so on. SQL is a standardized language, but it has also been customized by vendors who add or remove features. It is a declarative language; you declare what you want in a pseudo-sentence, instead of telling the computer which **for** loops to execute.

Database Management Systems

Database Management Systems (DBMSes) create database files in their own formats and execute SQL (typically) upon those files in order to behave like a database, according to our aforementioned desired features. The market of DBMSes is quite di-

verse.. Some are free; some are exhorbitantly expensive. Some are well engineered; some are hack jobs. Some are lightweight for reduced feature sets; some are cranked up to 11. And everything in between.

We will see two types of DBMSes that are appropriate for powering websites: **embedded databases** and **database servers**. We will see **SQLite** as our preferred embedded database and **PostgreSQL** as our preferred database server.

DBMS APIs

DBMSes typically expose a few different types of interfaces. There is a command-line interface, where you can type SQL commands directly and other program-specific commands, and there are APIs. These APIs are libraries written in a programming langauge in order to expose function calls, et. al., which provide interactivity with the database. There are C APIs, C++ APIs (though mostly just wrappers of the C APIs), and PHP APIs, to name a few.

Using the PHP APIs for these DBMSes, a website can interact

with a database. For the most part, these look like regular function calls which pass in strings of SQL code. We will see more details when we look at specific DBMS products in the coming lessons.