ECE356: Database Systems Winter 2018

Lecture 1 — Database Systems

Jeff Zarnett

Databases and Database Systems

Databases and database systems are everywhere. It is safe to say that it would not be possible you to read this pdf (or attend the lecture in which it is used) without the involvement of databases. They store data in an orderly fashion and allow for its retrieval at a later date. Thus a database keeps track of what students are attending the university now, who is enrolled in what course, what PDF files are associated with what term's offering of the course, and so on. The more traditional examples relate to everyday life for a common person: banking, airline reservations, e-commerce websites...

A database is by no means the only way to store data; data may be stored in any format one wishes. Professors, for example, are notorious for storing their data in large unorganized piles of paper in their offices. Simple text files are often used to record data. A surprising amount of the world operates on the basis of Microsoft Excel spreadsheets. This is not because of the mathematical functions or intelligence of Excel but rather the gridlines. Seriously.

For small amounts of data it may not matter very much that the data is not structured and not organized. For larger amounts of data, this quickly becomes impractical. Some sort of structure is needed and the first approach to dealing with this was to use files. Programs stored data in files and manipulated those files. Suppose a simple example from [SKS11], about keeping track of students at university.

Assuming there are files for students, instructors, courses, et cetera, then we need programs (or functionality of a single program) to add students, add instructors, add courses, register students for courses, generate class rosters, assign grades to students, generate transcripts... As time goes on, new functionality is added and the functionality is extended and changed.

Files for data storage have some serious drawbacks:

- **Data Redundancy:** The same data may be duplicated (appear in more than one location in the files).
- **Data Inconsistency:** Copies of duplicate data may not agree (e.g., an address update for a person may not be reflected in all copies).
- **Data Isolation:** Data can be spread over lots of files and hard to get together to find the information you want.
- **Integrity Problems:** There are real world rules that need to be respected, such as the pieces of inventory in stock cannot become negative; files don't support rules.
- **Atomicity Problems:** Updates to data should be atomic: succeed completely or it should be as if it did not happen at all. File systems may, or may not provide guarantees about writing to them.
- **Concurrent-Access Problems:** If two operations take place on a file at the same time, they may, or may not produce the expected outcome (remember earlier courses about concurrency?)
- **Security Problems:** It is hard to restrict access to just a subset of data: a professor should be able to only update the grades of students taking her class in the current term, for example...

The database, or more specifically, a database management system (DBMS) is a solution to these problems. Of course, a database, if poorly designed, will not fix all of those problems. It is certainly possible to have redundant

or inconsistent data if the data storage is poorly designed. It is similarly possible to fail to introduce the rules to ensure integrity and suffer as the logical rules are violated.

There are three levels of abstraction we are interested in when discussing a database, from least abstraction to most [SKS11]:

1. Physical Level – How the data is actually stored.
2. Logical Level – How the data is structured (organized).
3. View Level – A virtual structure based on how we would like to see the data.

When examining the physical level, it matters how the data is stored (usually on disk), including the low level data structures to contain the data. Database administrators may care about this data and it may be necessary to manipulate the physical storage configuration to get better performance, But for the most part, application programmers are unaware of this level; they focus on the next level.

The logical level tells us what data is in the database and what the relationships are between those pieces of data. Because of the abstraction, when thinking about the database, there is no need to be aware of the physical level data structures or even their location. At this level data that belongs together is grouped together, in the same way that we might pack data together in a structure or class.

At the view level, we are looking at only a part of the database. There are often many views for the same database, such as one for students and another for instructors and yet a third for university administrators. We will create lots of different views of the same data based on the needs of the users (or database administrators who want to check on things).

A database is typically accessed via a database server. Thus we see the client-server model in use: an application program then accesses the database through some request interface and receives answers. There also exist client programs that allow you to send commands to the database directly and get responses. This is one way that we might see what the current state of the database is. We can also issue statements that change the data directly, although caution must obviously be used to ensure correctness; an incorrect statement can do a lot of damage and lead, perhaps, to unrecoverable data loss. Soon enough, we will begin to learn the format and syntax for issuing database commands.

Schemas & Instances

A database *schema* defines the structure of the database, including what data is to be stored in what format. An instance of a database reflects both the schema and the particular content of that database. If this analogy helps, you may think of a schema as being like a class file definition in an object-oriented programming language. Except the schema can define multiple types of object and the database instance includes instances of many different types.

A (logical) schema may look something like the example below, which represents some motor vehicle registration data. This is the sort of thing that ministries of transport keep on hand so they can identify vehicles and their owners and their license plates. Keep in mind that the design shown is a choice; there are other options for how it could be organized.

VEHICLE

VIN	`string(17) not null, primary key`
year	`integer not null`
make	`string(64) not null`
model	`string(64) not null`
license_plate_number	`string(8) not null`

LICENSE_PLATE

number	`string(8) not null, primary key`
expiry	`date`
owner_address_id	`integer not null`

OWNER_ADDRESS

id	`integer not null, primary key`
name	`string(64) not null`
street	`string(64) not null`
city	`string(32) not null`
province	`string(2) not null`
postal_code	`string(7) not null`

There are some immediate observations. The first is these three structures (vehicle, license plate, and owner address) are all intended to represent some analogue to a real life thing in the same way that objects in an object-oriented programming language do (or structures in a language like C). Each entity is broken down into some number of fields, and each field has a name and a type. Types have some meaning, so a year must be an integer, like "2016", and cannot be a string or contain a decimal. There are potentially restrictions on a type, such as the "not null" requirement. Field names must be relatively unique but do not have to be globally unique. There also appear to be data elements that share some names with other data elements or other entities. That is a subject we will return to later.

There are also some items of meta-data (data about data) in the shown schema. For example, `id` is shown as "primary key", which provides additional information and additional restrictions. There are also indexes[1] that are used in searches. They can be added, removed, and modified, without affecting the underlying data.

No information is shown here about how the physical schema looks; how is this data stored in a practical sense? From the logical view it is irrelevant. If we wish, we can move some frequently accessed data to a faster device (SSD?) without affecting the logical schema and therefore without affecting any program that relies on this data.

If we create a new database and apply this schema, we created a new instance but it is empty. This is the initial state of the database and each modification to the database creates a new state of the database. At any time, we can take a snapshot of the current database (export all its data). This is a common operation for taking backups, for example, of the data, or just to make a copy to use on another test system.

The most common type of modification would be to manipulate data in it: add an item (e.g., a new car is purchased and the data gets entered), modify an item (e.g., someone moves and their address needs to get updated), or delete an item (e.g., a license plate is retired from service). This kind of change does not modify the schema, but it does modify the data. If we attempt to modify the data in a way that is not consistent with the schema (e.g., try to put "ABC" in an integer field), that modification will be rejected. Nevertheless, care must be taken in what data updates are issued, otherwise we may overwrite or delete important data. An incorrectly-crafted update command can easily cause unintended damage such as setting all VINs to be the same value. Careful schema design, as we will see, will prevent that sort of error, such as introducing a requirement that VINs be unique. If that rule exists, the update is rejected. Design cannot prevent all such errors, unfortunately. There can be two people with the name "Thomas Anderson"[2], so no rule of unique names can be introduced, and an erroneous update command could set all names to exactly that even if the schema is designed perfectly.

Data modification is not the only kind of change, however, because the schema can and does change. Textbook authors like in [EN11] sometimes tell you that schema changes will be rare, but much depends on your application. If it is a stable application then schema changes are rare; if it is a startup company or an app with ever-expanding functionality then the the schema will change regularly. Schema changes are design changes, though, and they do not typically happen in the normal operation, but instead happen during version upgrades.

Modifications to the schema can be made in a way that does not affect the data or require changes to applications that access it. If a new field "country" is added to the address entity, then something that looks only for the name

[1] Actually, indices is the correct plural, but "indexes" is used very commonly in English
[2] Whoah.

will not be affected. Similarly, a new entity, unrelated to these, such as "Service Ontario locations" (places where drivers need to wait in agonizing lines to register vehicles, for example) can be created with no impact on the existing data.

Modifying a schema, can, of course, be dangerous. If we decide to remove a data element, such as the province from the address, then data may be lost. If we add another property, such as a country, by default it contains nothing – and if this property is to be defined as "not null" then a default value is needed.

The database server can, of course, have multiple instances, and multiple schemas[3] all of which are independent of one another. Two databases may use the same schema but contain completely different data. Two databases can begin with the same schema but evolve in different directions over time based on the changes that are made. A change in the schema may result in changes to some data (e.g., data is truncated when a field is made smaller) but does not have to (e.g., a field is made larger and the existing data is unchanged).

The entities (vehicle, license plate, owner address) themselves are only a part of the picture. There are also the relationships between these, which are partially reflected in the fields like `owner_address_id`. This takes us to our next subject, which is to examine the relational model in more detail.

References

[EN11] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 6th Edition*. Addison-Wesley, 2011.

[SKS11] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 6th Edition*. McGraw Hill, 2011.

[3] The plural of schema is schemata, but I think we know how this goes by now.