Introduction to Databases

Databases and database applications are everywhere. When the cashiers ring you at the supermarket, they are accessing a database; when you enroll in a class, that information is stored in a computer database; all the bills that come to you are generated by database applications.

This book introduces computer databases, mainly from the perspective of a database designer and software developer. We assume you will be writing database applications; that is, programs that access a database.

1 Basic Definitions

Most computer applications store and manipulate data, in one way or another. Initially, as people wrote different applications, each group of programmers would write routines to store data, usually in fies in a file system; these routines would be completely different for each application (chances are, you were asked to write a program that reads and writes records from a file, for your introductory programming classes).

As we gained more experience, we realized that it would make sense to standardize some of those routines, still at a file level, and we created libraries and programming languages with standard routines to access files, where each file contained records of the same type; languages like COBOL and RPG. These routines and languages were much better than starting from scratch, but were still limited.

Eventually, these routines grew into complex pieces of software, designed to store and manipulate large amounts of data, to control multi-user access and to provide for easy ways to produce ad-hoc reports; we now call this software a Database Management System.

When dealing with computer databases it is useful to distinguish between the actual database (the data) and the database applications (the programs that access that data). Actually, since database applications are so important, we have found it convenient to create pieces of software that deal with databases in general, and so allow us to create database applications easier; we call that piece of software a DataBase Management System (DBMS).

Computer Database: collection of data stored in a computer system.

More formally, we can define those concepts as follows:

- A **Computer Database (DB)** is a collection of data stored in a computer system. In this book, we assume all databases will be stored in a computer, so we don't distinguish between databases and computer databases.
- A **Database Management System (DBMS)** is a piece of software that is able to provide access to databases in general.
- A **Database Application** is a piece of software that provides value to a user and accesses one (or a small number of) specific databases.

Database Management System (DBMS): software that is able to provide access to databases in general.

Database Application: application designed to access one (or a small number of) specific databases.

Data: individual facts.

Although we need to distinguish these three terms so we can study them separately, it is extremely common for people to use the term 'database' to refer to each one of them. For most users of a database application, the application *is* the database, and many programmers refer to the DBMS, or to the computer on which the DBMS runs, as the database.

When designing database applications, it is useful to distinguish between **data**, that is the individual facts, and **information**, which is processed data (hopefully so as to be more useful to the users). For the most part, what will be input into a database application is raw data, but what we want to get out of it is information. Notice this is *not* a hard and crisp distinction. For many users, the data is also information, and what is information for one user may be raw data for another.

Information: processed data

For example, imagine a university, with several departments, which offers many classes. Data about individual students, which classes they take and what grades they get would be considered data, since it is what we put into the system. For the students, this is also information, since they want to know their grades. For the teachers, the individual grades are information, but also things like class GPA etc; for a department chair, which particular students are taking each course (raw data) may not be important, but only the number of students in each course; this number of students in each course may be considered raw data for the president of the university, who may be concerned just about the average number of students in a department's courses.

An important kind of data is **metadata**, that is data about other data. This includes computer metadata (the data type used, the number of bits or bytes per field etc), that is, metadata that is directly used by computer systems that manipulate the data, and human documentation (what the fields in the database actually mean, where does each piece of data comes from, who is authorized to change the definitions, etc).

Metadata: data about other data.

Another important distinction when thinking about databases, is between a database's **schema**, that is
instance, which is the data present in that particular database at a given moment in time. Obviously, the data *in* a database changes over time, while the schema changes very little, if at all.

Database Schema: Description of all possible values in a database, along with the constraints those values must

Formally:

- A **Database Schema** is a description of all the possible values in a database, along with the constraints those values must satisfy.
- A **database instance** is the set of values present in a particular database at a particular moment in time.

Database Instance: Set of values present in a particular database at a particular moment in time

Sometimes, we call a database's schema its **intension** and the particular instance its **extension**.

The basic operations a database application performs are to Create (or add), Read, Update and Delete data, which we can remember with the acronym CRUD. Most database applications do *not* provide ways to alter the database's schema within the application, just the extension of the database (the actual data inside the database). DBMSs, of course, provide ways to create new databases and to alter their schema, besides CRUD operations on data.

2 Database Applications vs File System Applications

Before Database Management Systems became popular, we had file-processing applications. These applications dealt with data, but didn't view a database as an integrated repository of data, just as a set of files. The data in a file was processed by a program and then dumped into another file. Of course, we still have many of those applications still being used (mostly in mainframes) and a few of them being written.

When you write a program that deals directly with files, that program needs to know the exact format of the file; this makes changes to the file format very difficult. Any change to the file format implies a change in the program, and also the conversion of all files from the previous format to the new format. And of course, a complete application would probably have several modules, many of which access the same file, and all of which need to change when the format of the file changes.

This need to change a program when the format of the files it access changes is called **program-data dependence**, and is a real problem for most file-processing applications. For big companies, where many applications need to access the same data, and those applications were under different departments, it lead to each department defining its own file formats, and made for very limited data sharing, plus large maintenance costs, since each change meant changing several programs (and hopefully testing them), and also required writing a conversion program, to convert old files to the new format.

So, given that program-data dependence is a big problem, we add one more abstraction layer, the DBMS, to achieve **program-data independence**[1]. The computer metadata for each file is stored with the file, and this new software layer can interpret that metadata and access the data in the file regardless of changes in the format. Over time, other functions were added to this layer, for increasing performance and facilitating the programmer's work.

program-data independence: separation of programs from the *format* of the data they use, so changes to the data format do not necessitate changes to the program.

The final step to the modern DBMS was the standardization of interfaces, specifically the use of SQL for accessing the databases, and converting the DBMS into a separate server program accessible over the network. Nowadays the DBMS provides not only program-data independence, but remote access and multi-user capabilities. DBMSs are also able to enforce many constraints on the data in a database, so each application can be sure the data satisfies *some* standards. Notice this does *not* guarantee the data is *correct*, just that is is not *obviously* wrong.

Another capability enabled by program data independence is the ability to write generic clients to access the data, and so having ad-hoc reports generated easily. Up to a point, this allows some non-programmers to write their own queries and reports, minimizing the need for custom report development.

Now, given this technical ability to have the data in one place, share it between multiple applications and enforcing constraints on the data, we can have an *organizational* decision to have a centralized database, with applications from all departments accessing *only* this database, and invest resources into making this data of high quality. Notice that we may still decide to have each department have its own version of the data, in slightly different formats and with some disagreements among the databases.

The decision to centralize the data is an organizational decision, and many times even a *political* one; but the technological capabilities of modern DBMSs allow for greater centralization and control of the data.

Given this, we can list the following advantages of database applications (as compared to direct file system applications), noticing that, except for program-data independence, the other

1 A better name would be **program-metadata independence**, since the program is independent of the format of the data, that is, the metadata; however, program-data independence is the commonly used term.

advantages are *potential*, and depend not just on database technology but also on organizational issues.

1. Program-data Independence
2. Minimized Data Redundancy
3. Improved Data Consistency, Standards and Quality
4. Improved Data Sharing
5. Improved Data accessibility and Decision support (ad-hoc queries etc)
6. Reduced program maintenance

The main *disadvantage* of the database approach is the need for specialized software, the DBMS. This means people need to be trained for it (but that's what you're doing right now), and probably the need to have a **database administrator (DBA)**, that takes care of the DBMS and the databases managed by it. It used to be that installing and administering a DBMS was a very specialized task, but nowadays it is relatively simple, although, of course, and experienced DBA will probably be able to achieve much better performance and reliability from a given DBMS. The DBMS may also need specialized backup procedures.

3 Database Sizes

Most database applications share common characteristics, which is why we classify those applications as database applications; however, depending on a number of factors, most notably the number of users, the size of the database and the applications dependability requirements, we may need to use different technologies, so it is often convenient to classify the applications according to their size.

Notice that as technology changes, it may become convenient to classify in a different way or to adjust the categories. However, in 2008, we find convenient to classify the applications as follows:

- **Personal Database Applications –** These will be accessed by one person only. Nowadays this usually means palmtops and mobile devices, since on PCs we develop most databases for the possibility of having more than one user at a time. The database itself will usually be very small and the application doesn't need network connectivity, so the DBMS may be embedded in the application, rather than use client-server technology.
- **Workgroup Database Application –** These are databases accessed by a relatively small number of people at a time (say less than 100), and most of them on the local network. Here we normally want client-server technology, with a DBMS server, but we do not need specialized hardware, and almost any DBMS will do.
- **Enterprise database applications –** Enterprise applications are those that, when not working, bring down the whole company; therefore, we need a special emphasis on reliability. We usually need specialized hardware (real servers, oftentimes Unix or Mainframes), and we oftentimes have a large number of users, and many of them accessing the database remotely.
- **Internet Database Applications –** Here we have a very large number of users, but they seldom access the application. All of the users come remotely, and their client is a web browser.

Keep in mind these categories are fuzzy, and they are changing. Technology has been converging rapidly. Advanced DBMSs have become cheaper and easier to use, so we use them even for small applications; we are becoming more adept at developing web-based applications, so nowadays we develop many applications as if they were internet applications, even though they will be used as workgroup or even personal applications.

4 Clients, Servers and Tiers

A server is just a program that is designed to be accessed over a network, rather than directly by a user on the same computer. Modern DBMSs provide a client-server architecture. The DBMS is accessed over a network; clients send SQL commands (which may include commands to modify the data in the database) and receive data from the database.

Modern DBMS servers can be accessed by many clients. Most DBMSs provide at least one generic client able to process typed SQL statements, plus at least one generic GUI client, able to do most database operations.

As far as the DBMS is concerned, most database applications are just another client; although each database application is tailored to a particular database.

In the traditional client-server architecture, the client application is a full application running on the client machine. This means that all the clients need to be able to run the application (so, if you write your application for Linux, then all your client machines need to be running Linux, and have all the libraries needed for your application). This is an initial problem because it requires a more or less homogeneous environment; however the worst problem is updating the clients. If you need to update the client application, then all the clients need to be updated, or you need to deal with the fact that many completely different clients are accessing your database.

This problem with the traditional client-server architecture started a push towards clients that did less and less, so called lean clients, as opposed to fat clients, with the bulk of the business logic done in another server, an application server. With the popularity of the internet, a web browser became the obvious lean client target.

This kind of application is called a three-tier application (a tier is just a fancy name for a layer of software, where the layers can be connected through a network rather than on the same computer); we have the client tier, which is just a web browser, connecting to an application server; which is usually a web server able to run programs; this was done initially through CGI and later by embedding interpreters in the web server, as is usually done in PHP and ASP, or by embedding a web server as one module of the application server, as is commonly done with Java Servlets.

tier: a software layer, where the layers can be accessed through a network rather than being on the same computer.

5 The Modern Three-Schema Database Architecture

Previously we said that a schema is a description of the database. Different people need different descriptions of the database (that is, different schemas), at different levels of abstraction. Of course, all these descriptions should agree, since they are describing the same database.

We usually find it convenient to have three different schemas:

conceptual schema: abstract description of the database, independent of database technology.

- A **conceptual schema**, which is an abstract description of the database, independent of database technology (usually as an entity-relationship model). The conceptual schema is used by the database designer to communicate with the users who are providing the requirements for the application.

- A **logical schema**, which is a detailed description of the database, for a particular database technology (usually relational) in which the application will be implemented. The logical schema is usually the level the DBMS deals with, but it abstracts most of the details about how the data will actually be stored in the computer. This is the level at which programmers usually communicate.

logical schema: detailed description of the database for a particular database technology (usually relational).

- A **physical schema** is a description of the database including all the details about how it will be stored in a computer. This level is what the Database administrator sees.

physical schema: description of the database including all the details about how it will be stored in a computer.

Basically, to implement the database, we need a lot of very detailed information that is not useful at other levels. We need to know the specific format of each field, and how are the fields organized in a record (in which order, do we have gaps, etc). We also need to know how are the records organized in the database (do we store them in the order they come in, or in a particular order ? Do we leave any space between the records ? Do we store all records of the same kind together ? etc). Moreover, we need to know how to map the database objects to operating system objects and to the actual hardware (Is the database stored in an operating system file or in a raw partition ? which disk ? which file or partition ? etc).

All of these issues are important and will greatly affect performance; however, these issues do *not* affect the semantics of the database. It is convenient for the programmer writing database applications to not worry about this issues, but to have a more abstract view of the database. Relational DBMSs (the most common kind by far) provide a view as a set of tables, each table containing records of the same kind. As far as programmers are concerned, they are just accessing records in a table. So this tables are the logical schema.

Modern relational DBMSs provide ways to change the physical schema without changing the logical schema at all; in fact, this is the way they achieve program-data independence.

Although the logical schema is abstract enough for the programmers, it is still too detailed for end user, who will be providing the requirements. It is common for the analyst to use a graphical notation for communicating with the end users. The most common notation for database applications is the Entity Relationship Diagram, which provides a very abstract representation of data, in terms of entities (roughly corresponding to classes) and their relationships. This model is used during analysis or requirements gathering, and is later translated into a logical schema.

Notice that, depending on the application we may not need all of these schemas. For example, if the users are also programmers, or are familiar with relational databases, we may not need to have a separate conceptual schema. In this book we will cover how and why to do each of these schemas, and will assume that you are doing all of them.

Another important distinction is between the **internal schema** and **external schemas** which may exist. The internal schema is the view of the whole database structure, while external schemas are subsets of the internal schema for particular groups of users. The external schemas are also called **user views**; and they are usually at the logical level.

internal schema: structure of the complete database.

external schema (user view): subset of the internal schema for a particular group of users.

6 Software Development Process for Database Applications

When creating applications, we ought to follow some process. A simplified view of the software development process would contain the following steps:

1. **Analysis** (or requirements gathering), where we help the users decide what exactly the application needs to do; the most important database activity here is conceptual modeling, in which we create a conceptual schema for the database.
2. **Design**, where we decide how we will structure the software. For databases, we separate the logical design (where we transform the conceptual schema into a logical one) from the physical design (where we add necessary details to the logical schema to transform it into a physical schema). Since DBMSs allow us to change the physical schema without changing the logical one, we can let the DBMS select default choices, and defer many physical design decisions until implementation or even maintenance.
3. **Implementation**, where we actually build the software. In this step we need to actually implement our database design in a particular DBMS, which may require small adjustments to the design. As we implement more of the application and can see the performance implications of our physical design decisions, we may adjust the physical design.
4. **Maintenance**, which involves changing the software to adjust to changing requirements or to fix any errors. Performance problems will oftentimes be fixed by adjusting the physical design. Notice that adding fields to a database may be a very involved task, since it may require capturing data for many different records.

Notice this is a simplistic view of software process. It simplifies the process in two big ways:

1. It assumes the steps are actually separate. This is reasonable for studying the processes, but far from true. in real life, it is very hard to do analysis without suggesting a design, and it is hard to delineate exactly where analysis ends and design starts. Also, nowadays the implementation step involves much detailed design.
2. It assumes the steps are performed sequentially. Again, this is not true. In real life, there is at least some (and usually a lot) looping back from one state back to a previous one.

There is a wide variety of software development processes, but we can roughly classify them into two broad categories:

1. **Traditional Waterfall Processes**, which assume the actual process should be as close as possible to the simplified view; they usually put a lot of emphasis in the requirements process, and have relatively heavy bureaucracy to guarantee the process is followed and there are very few loops back. Well done, these processes have a high overhead but maximize the chances of having an application that satisfies the original requirements. They are well-suited for big projects, or software developed under contracts. They are not suited for short projects, or for projects where the requirements are not well understood or are likely to change.
2. **Iterative processes**, which assume the requirements *will* change over time, and so it is pointless to try to follow the traditional process. They usually involve several cycles of the process, each one of them producing a piece of software with partial functionality, with the amount of functionality increasing over time. Recently iterative processes (at least some of them) have been identified as agile. One advantage of these processes is that they usually provide partial functionality early on, which makes them well-suited for short projects. The worst problem with these processes is that they can easily degenerate into code-and-fix, with little though put into analysis and design, and consequently waste lots of effort.

In this book, we will use the simplistic view when discussing software processes, focusing mostly on database applications. Notice we are not endorsing waterfall models for actual use, we just find it a better framework for initial understanding and discussion of basic software processes.