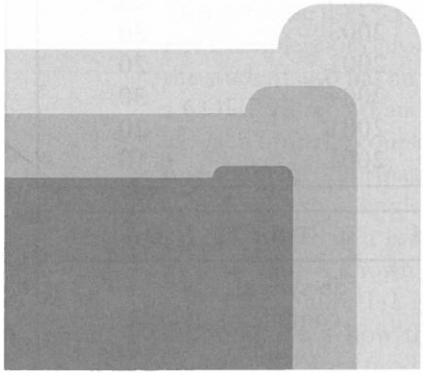
1 Introduction: Why a Database System?



Database systems are a big deal in the computer industry. Some database systems (such as Oracle) are enormously complex, and typically run on large, high-end machines. Most database systems have a special user, called the *database administrator* (or *DBA*), who is responsible for its smooth operation. The DBA in a large database installation is very often a dedicated full-time employee; some installations even have several full-time administrators. This chapter examines the features a database system is expected to have, to better understand why it takes so many resources to implement these features.

1.1 Databases and Database Systems

A *database* is a collection of data stored on a computer.

The data in a database is typically organized into *records,* such as employee records, medical records, sales records, and so on. Figure 1-1 depicts a database that holds information about students in a university and the courses they have taken. Our university database will be used as a running example throughout the book.

The database of Figure 1-1 contains five types of records:

- There is a STUDENT record for each student that has attended the university. Each record contains the student's ID number, name, graduation year, and ID of the student's major department.

STUDENT	SId	SName	GradYear	MajorId
	1	joe	2004	10
	2	amy	2004	20
	3	max	2005	10
	4	sue	2005	20
	5	bob	2003	30
	6	kim	2001	20
	7	art	2004	30
	8	pat	2001	20
	9	lee	2004	10

DEPT	DId	DName
	10	compsci
	20	math
	30	drama

COURSE	CId	Title	DeptId
	12	db systems	10
	22	compilers	10
	32	calculus	20
	42	algebra	20
	52	acting	30
	62	elocution	30

SECTION	SectId	CourseId	Prof	YearOffered
	13	12	turing	2004
	23	12	turing	2005
	33	32	newton	2000
	43	32	einstein	2001
	53	62	brando	2001

ENROLL	EId	StudentId	SectionId	Grade
	14	1	13	A
	24	1	43	C
	34	2	43	B+
	44	4	33	B
	54	4	53	A
	64	6	53	A

Figure 1-1
Some records for a university database

- There is a DEPT record for each department in the university. Each record contains the department's ID number and name.
- There is a COURSE record for each course offered by the university. Each record contains the course's ID number, title, and the ID of the department that offers it.
- There is a SECTION record for each section of a course that has ever been given. Each record contains the section's ID number, the year the section was offered, the ID of the course, and the professor teaching that section.
- There is an ENROLL record for each course taken by a student. Each record contains the enrollment ID number, the ID numbers of the student and the section of the course taken, and the grade the student received for the course.

This database uses several small records to store information about one real-world object. For example, not only does each student in the university have a STUDENT record, but the student also has an ENROLL record for each course taken. Similarly, each course has a COURSE record and several SECTION records—there will be one SECTION record for each time the course has been offered. There are, of course, many other ways to structure the same information into records. For example, we could put all information about a student into a single large record; that record would contain the values from STUDENT as well as a list of ENROLL subrecords. The process of deciding how to structure the data is known as *designing* the database, and is the subject of Chapter 3.

Now, Figure 1-1 is just a conceptual picture of some records. It does not indicate anything about how the records are stored or how they are accessed. There are many available software products, called *database systems,* which provide an extensive set of features for managing records.

> A *database system* is software that manages the records in a database.

What does it mean to "manage" records? What features must a database system have, and which features are optional? The following five requirements are fundamental:

- *Databases must be persistent.* Otherwise, the records would disappear as soon as the computer is turned off.
- *Databases can be very large.* The database in Figure 1-1 contains only 29 records, which is ridiculously small. It is not unusual for a database to contain millions (or even billions) of records.
- *Databases get shared.* Many databases, such as our university database, are intended to be shared by multiple concurrent users.
- *Databases must be kept accurate.* If users cannot trust the contents of a database, it becomes useless and worthless.
- *Databases must be usable.* If users are not able to easily get at the data they want, their productivity will suffer and they will clamor for a different product.

Some database systems are incredibly sophisticated. Most have a special user, called the *database administrator* (or *DBA*), who is responsible for its smooth operation. The

DBA in a large database installation is very often a dedicated full-time employee; some installations even have several full-time administrators.

> A *database administrator* (or *DBA*) is the user who is responsible for the smooth operation of the database system.

Why does it take so much effort and so many resources to manage a database? The remaining sections in this chapter examine this question. We will see how each requirement forces the database system to contain increasingly more features, resulting in more complexity than we might have expected.

1.2 Record Storage

A common way to make a database persistent is to store its records in files. The simplest and most straightforward approach is for a database system to store records in text files, one file per record type; each record could be a line of text, with its values separated by tabs. Figure 1-2 depicts the beginning of the text file for the STUDENT records.

This approach has the advantage that the database system has to do very little; a user could examine and modify the files with a text editor. However, this approach is unsatisfactory for two reasons:

- Updating the file takes too much time.
- Searching the file takes too much time.

Large text files cannot be updated easily. For example, suppose a user deletes Joe's record from the text file. The database system would have no choice but to rewrite the file beginning at Amy's record, moving each succeeding record to the left. Although the time required to rewrite a small file is negligible, rewriting a one gigabyte file could easily take several minutes, which is unacceptably long. A database system needs to be much more clever about how it stores records, so that updates to the file require only small, local rewrites.

The only way to search for a record in a text file is to scan the file sequentially. Sequential scanning is very inefficient. You probably know of many in-memory data structures, such as trees and hash tables, which enable fast searching. A database system needs to implement its files using analogous data structures. In fact, a record type might have

1 [TAB] j o e [TAB] 2 0 0 4 [TAB] 1 0 [RET] 2 [TAB] a m y [TAB] 2 0 0 4 [TAB] 2 0 [RET] ...

Figure 1-2
Implementing the STUDENT records in a text file

several auxiliary files associated with it, each of which facilitates a particular search (e.g., on student name, graduation year, or major). These auxiliary files are called *indexes*.

> An *index* is an auxiliary file that allows the database system to search the primary data file more efficiently.

1.3 Multi-User Access

When many users share a database, there is a good chance that they will be accessing some of the data files concurrently. Concurrency is a good thing, because each user gets to be served quickly, without having to wait for the other users to finish. But too much concurrency is bad, because it can cause the database to become inaccurate. For example, consider a travel-planning database. Suppose that two users try to reserve a seat on a flight that has 40 seats remaining. If both users concurrently read the same flight record, they both will see the 40 available seats. They both then modify the record so that the flight now has 39 available seats. Oops! Two seats have been reserved, but only one reservation has been recorded in the database.

A solution to this problem is to limit concurrency. The database system should allow the first user to read the flight record (with its 40 available seats) and block the second user until the first user finishes. When the second user resumes, it will see 39 available seats and modify it to 38, as it should. In general, a database system must be able to detect when a user is about to perform an action that conflicts with an action of another user, and then (and only then) block that user from executing until the first user has finished.

Users may need to undo updates they have made. For example, suppose that a user has searched the travel-planning database for a trip to Madrid, and found a date for which there is both an available flight and a hotel with a vacancy. Now suppose that the user reserves the flight; but while the reservation process is occurring, all of the hotels for that date fill up. In this case, the user may need to undo the flight reservation and try for a different date.

An update that is undoable should not be visible to the other users of the database. Otherwise, another user may see the update, think that the data is "real," and make a decision based on it. The database system must therefore provide a user with the ability to specify when his changes are permanent; we say that the user *commits* his changes. Once a user commits his changes, they become visible and cannot be undone.

Authentication and authorization also become important in a multi-user environment. The records in a database typically have different privacy requirements. For example, in the university database, the dean's office might be authorized to view student grades, whereas the admissions office and the dining hall workers might not. A database system must therefore be able to identify its users via usernames and passwords, and authenticate them via some sort of login mechanism. The system must also have a way for the creator of a record to specify which users can perform which operations on it; these specifications are called *privileges*.

1.4 Memory Management

Databases need to be stored in persistent memory, such as disk drives or flash drives. Flash drives are about 100 times faster than disk drives, but are also significantly more expensive. Typical access times are about six milliseconds for disk and 60 microseconds for flash. However, both of these times are orders of magnitude slower than main memory (or RAM), which has access times of about 60 nanoseconds. That is, RAM is about 1000 times faster than flash and 100,000 times faster than disk.

To see the effect of this performance difference and the consequent problems faced by a database system, consider the following analogy. Suppose that your pocket is RAM, and that you have a piece of data (say, a picture) in your pocket. Suppose it takes you one second to retrieve the picture from your pocket. To get the picture from the "flash drive" would require 1000 seconds, which is about 17 minutes. That is enough time to walk to the school library, wait in a very long line, get the picture, and walk back. (Using an analogous "disk drive" would require 100,000 seconds, which is over 24 hours!)

Database support for concurrency and reliability slows things down even more. If someone else is using the data you want, then you may be forced to wait until the data is released. In our analogy, this corresponds to arriving at the library and discovering that the picture you want is checked out. So you have to wait (possibly another 17 minutes or more) until it is returned. And if you are planning on making changes to the picture, then you will also have to wait (for possibly another 17 minutes or more) while the library makes a backup copy of it.

In other words, a database system is faced with the following conundrum: It must manage *more* data than main memory systems, using *slower* devices, with *multiple people* fighting over access to the data, and make it *completely recoverable,* all the while maintaining a reasonable response time.

A large part of the solution to this conundrum is to use *caching.* Whenever the database system needs to process a record, it loads it into RAM and keeps it there for as long as possible. Main memory will thus contain the portion of the database that is currently in use. All reading and writing occurs in RAM. This strategy has the advantage that fast main memory is used instead of slow persistent memory, but has the disadvantage that the persistent version of the database can become out of date. The database system needs to implement techniques for keeping the persistent version of the database synchronized with the RAM version, even in the face of a system crash (when the contents of RAM is destroyed).

1.5 Data Models and Schemas

So far, we have seen two different ways of expressing the university database:

- as several collections of records, as in Figure 1-1;
- as several files, where each record is stored in a particular representation (as in Figure 1-2) and index files are used for efficiency.

Each of these ways can be specified as a *schema* in a *data model.*

> A *data model* is a framework for describing the structure of databases.
> A *schema* is the structure of a particular database.

The first bullet point above corresponds to the *relational data model.* Schemas in this model are expressed in terms of tables of records. The second bullet point corresponds to a *file-system data model.* Schemas in this model are expressed in terms of files of records. The main difference between these models is their level of abstraction.

A file-system schema specifies the physical representation of the records and their values. For example, a file-system schema for the university database might specify student records as follows:

- Student records are stored in the text file "student.txt."
- There is one record per line.
- Each record contains four values, separated by tabs, denoting the student ID, name, graduation year, and major ID.

Programs that read (and write to) the file are responsible for understanding and decoding this representation.

A relational schema, on the other hand, only specifies the fields of each table and their types. The relational schema for the university database would specify student records as follows:

- The records are in a table named STUDENT.
- Each record contains four fields: an integer *SId,* a string *SName,* and integers *GradYear* and *MajorId.*

Users access a table in terms of its records and fields: They can insert new records into a table, and retrieve, delete, or modify all records satisfying a specified predicate.

For example, consider the following two student schemas, and suppose we want to retrieve the names of all students who graduated in 1997. Figure 1-3(a) gives the Java code corresponding to the file-system model, and Figure 1-3(b) gives the equivalent code using SQL, which is the standard language of the relational model. We shall discuss SQL fully in Chapter 4, but for now this SQL statement should be easy to decipher.

The difference between these two code fragments is striking. Most of the Java code deals with decoding the file; that is, reading each record from the file and splitting it into an array of values to be examined. The SQL code, on the other hand, only specifies the values to be extracted from the table; it says nothing about how to retrieve them.

These two models are clearly at different levels of abstraction. The relational model is called a *conceptual model,* because its schemas are specified and manipulated conceptually, without any knowledge of (and without caring) how it is to be implemented. The file-system model is called a *physical model,* because its schemas are specified and manipulated in terms of a specific implementation.

```
public static List<String> getStudents1997() {
      List<String> result = new ArrayList<String>();
      FileReader rdr = new FileReader("students.txt");
      BufferedReader br = new BufferedReader(rdr);
      String line = br.readLine();
      while (line != null) {
            String[] vals = line.split("\t");
            String gradyear = vals[2];
            if (gradyear.equals("1997"))
                  result.add(vals[1]);
            line = br.readLine();
      }
      return result;
}
```

(a) Using a file system model

```
select SName from STUDENT where GradYear = 1997
```

(b) Using the relational model

Figure 1-3
Two ways to retrieve the name of students graduating in 1997

A *physical schema* describes how the database is implemented.
A *conceptual schema* describes what the data "is."

Conceptual schemas are much easier to understand and manipulate than physical schemas, because they omit all of the implementation details.

The relational model is by far the most prevalent conceptual data model today, and is the focus of this book. However, database systems have been built using other conceptual data models. For example:

- *Graph-based models* treat the database as linked collections of records, similar to the class diagrams of Chapter 3.
- *Object-oriented models* treat the database as a collection of objects, instead of records. An object can contain a reference to another object, and can have its own object-specific methods defined for it. The Java Persistence Architecture of Chapter 9 allows a database to be viewed in this way.
- *Hierarchical models* allow records to have a set of subrecords, so that each record looks like a tree of values. The XML data format of Chapter 10 has this conceptual structure.

1.6 Physical Data Independence

A conceptual schema is certainly nicer to use than a physical schema. But operations on a conceptual schema need to get implemented somehow. That responsibility falls squarely on the shoulders of the database system.

The database system translates operations on a conceptual schema into operations on a physical schema. The situation is analogous to that of a compiler, which translates programming language source code into machine language code. To perform this translation, the database system must know the correspondence between the physical and conceptual schemas. For example, it should know the name of the file containing each table's records, how those records are stored in the file, and how the fields are stored in each record. This information is stored in the *database catalog*.

> The *database catalog* contains descriptions of the physical and conceptual schemas.

Given an SQL query, the database system uses its catalog to generate equivalent file-based code, similar to the way that the code of Figure 1-3(a) might be generated from the query of Figure 1-3(b). This translation process enables *physical data independence*.

> A database system supports *physical data independence* if users do not need to interact with the database system at the physical level.

Physical data independence provides three important benefits:

- ease of use;
- query optimization; and
- isolation from changes to the physical schema.

The first benefit is the convenience and ease of use that result from not needing to be concerned with implementation details. This ease of use makes it possible for nearly anyone to use a database system. Users do not need to be programmers or understand storage details. They can just indicate what result they want and let the database system make all of the decisions.

The second benefit is the possibility of automatic optimization. Consider again Figure 1-3. The code of part (a) is a fairly straightforward implementation of the SQL query, but it might not be the most efficient. For example, if there is an index file that allows students to be looked up by graduation year, then using that file might be better than sequentially searching the *student.txt* file. In general, an SQL query can have very many implementations, with widely varying execution costs. It is much too difficult and time consuming for a user to manually determine the best implementation of the query. It is better if the database system can look for the best implementation itself. That is, the database system can (and should) contain a *query optimization* component whose job is to determine the best implementation of a conceptual query.

The third benefit is that changes to the physical implementation do not affect the user. For example, suppose that the dean's office runs a query at the end of each academic year to find the senior having the highest GPA. Now suppose that the database files were recently restructured to be more efficient, with a different file organization and additional indexes. If the query were written using the file system model, then it would no longer work and the dean's office would have to rewrite it. But if the query is written against the conceptual model, then no revision is necessary. Instead, the database system will automatically translate the conceptual query to a different file system query, without any user involvement. In fact, the dean's office will have no idea that the database has been restructured; all that they will notice is that their query runs faster.

1.7 Logical Data Independence

Every database has a physical schema and a conceptual schema. The conceptual schema describes what data is in the database; the physical schema describes how that data is stored. Although the conceptual schema is much easier to use than the physical schema, it is still not especially user-friendly.

The problem is that the conceptual schema describes what the data is, but not how it is used. For example, consider again the university database. Suppose that the dean's office constantly deals with student transcripts. They would really appreciate being able to query the following two tables:

```
STUDENT_INFO (SId, SName, GPA, NumCoursesPassed,
              NumCoursesFailed)
STUDENT_COURSES (SId, YearOffered, CourseTitle, Prof, Grade)
```

The table STUDENT_INFO contains a record for each student, whose values summarize the student's academic performance. The STUDENT_COURSES table contains a record for each enrollment, and lists the relevant information about the course taken.

Similarly, the faculty in the university may be interested in having a table CLASS_LIST that lists the enrollments for their current classes, and the registrar's office may want a table that lists the current class schedule.

The set of tables personalized for a particular user is called the user's *external schema.*

Figure 1-4 depicts the relationships among the three kinds of schema in a database. Such a database system is said to have a *three-level architecture.* The arrows indicate the translation from one schema to another. For example, a query on an external schema will be automatically translated to a query on the conceptual schema, which will then be automatically translated to a query on the physical schema.

A database that supports external schemas is said to have *logical data independence.*

> A database system supports *logical data independence* if users can be given their own external schema.

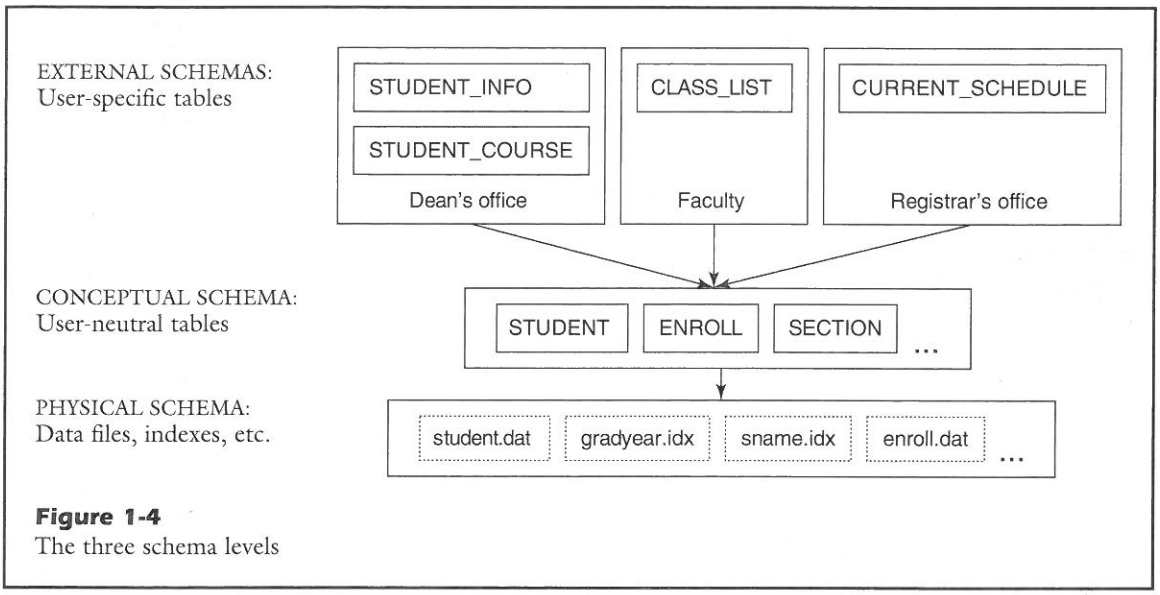


Figure 1-4
The three schema levels

Logical data independence provides three benefits:

- customized external schemas,
- isolation from changes to the conceptual schema, and
- better security.

The first benefit is that an external schema can be thought of as a user's customized interface to the database. Users see the information they need in the form that they need it, and they don't see the information they don't need. Moreover, if a user suddenly requires access to additional data, the database administrator can respond by simply modifying the user's external schema. Neither the conceptual schema nor any other external schemas are affected.

The second benefit is that users are not affected by changes to the conceptual schema. For example, suppose a database administrator restructures the conceptual schema by splitting the STUDENT table into two tables: CURRENT_STUDENT and ALUMNI. The administrator will also modify the definition of the external schemas that access STUDENT so that they access the new tables instead. Because users interact with student records only via their external schemas, they will be oblivious to this change.

The third benefit is that external schemas can be used to hide sensitive data from unauthorized users. Suppose that a user receives privileges only on the tables in his external schema. Then the user will not be able to access any of the tables in other external schemas, nor will the user be able to access any table in the conceptual schema. Consequently, the users will only be able to see the values they deserve to see.

The database system is responsible for translating operations on the external schema to operations on the conceptual schema. Relational databases perform this translation via *view definitions*. In particular, each external table is defined as a query

over the conceptual tables. When a user mentions an external table in a query, the database produces a corresponding query over the conceptual tables by "merging" the external table's definition query with the user query. View definitions will be covered in Section 4.5, and the merging process will be explained in Section 19.3.

1.8 Chapter Summary

- A *database* is a collection of data stored on a computer. The data in a database is typically organized into *records*.
- A *database system* is software that manages the records in a database.
- A *database administrator* (or *DBA*) is a person who is responsible for the smooth operation of a database system.
- A *data model* is a framework for describing the structure of databases. The relational model is an example of a *conceptual model*. The file-system model is an example of a *physical model*.
- A *schema* is the structure of a particular database. A *physical schema* describes how the database is implemented. A *conceptual schema* describes what the data "is." An *external schema* describes how a particular user wishes to view the data. A database system that supports all three kinds of schema is said to have a *three-level architecture*.
- A database system supports *physical data independence* if users do not interact with the database system at the physical level. Physical data independence has three benefits:
  - The database is easier to use, because implementation details are hidden.
  - Queries can be optimized automatically by the system.
  - The user is isolated from changes to the physical schema.
- A database system supports *logical data independence* if users can be given their own external schema. Logical data independence has three benefits:
  - Each user gets a customized schema.
  - The user is isolated from changes to the conceptual schema.
  - Users receive privileges on their schema only, which keeps them from accessing unauthorized data.
- A database system must be able to handle large shared databases, storing its data on slow persistent memory. It must provide a high-level interface to its data, and ensure data accuracy in the face of conflicting user updates and system crashes. Database systems meet these requirements by having the following features:
  - the ability to store records in a file in a format that can be accessed more efficiently than the file system typically allows;
  - complex algorithms for indexing data in files, to support fast access;
  - the ability to handle concurrent accesses from multiple users over a network, blocking users when necessary;

- support for committing and rolling back changes;
- the ability to authenticate users and prevent them from accessing unauthorized data;
- the ability to cache database records in main memory and to manage the synchronization between the persistent and main-memory versions of the database, restoring the database to a reasonable state if the system crashes;
- the ability to specify external tables via views;
- a language compiler/interpreter, for translating user queries on conceptual tables to executable code on physical files;
- query optimization strategies, for transforming inefficient queries into more efficient ones.

These features add up to a pretty sophisticated system. Parts 1 and 2 of this book focus on how to use these features. Parts 3 and 4 focus on how these features are implemented.

1.9 Suggested Reading

Database systems have undergone dramatic changes over the years. A good account of these changes can be found in Chapter 6 of *National Research Council 1999* (available online at *www.nap.edu/readingroom/books/far/ch6.html*), and in Haigh [2006]. The wikipedia entry at *http://en.wikipedia.org/wiki/Database_management_system* is also interesting.

1.10 Exercises

CONCEPTUAL EXERCISES

1.1 Suppose that an organization needs to manage a relatively small number of shared records (say, 100 or so).

a) Would it make sense to use a commercial database system to manage these records?

b) What features of a database system would not be required?

c) Would it be reasonable to use a spreadsheet to store these records? What are the potential problems?

1.2 Suppose you want to store a large amount of personal data in a database. What features of a database system wouldn't you need?

1.3 Consider some data that you typically manage without a database system (such as a shopping list, address book, checking account info, etc.).

a) How large would the data have to get before you would break down and store it in a database system?

b) What changes to how you use the data would make it worthwhile to use a database system?

1.4 Choose an operating system that you are familiar with.

- **a)** How does the OS support concurrent access to files?
- **b)** How does the OS support file authorization and privileges?
- **c)** Is there a user that corresponds to a DBA? Why is that user necessary?

1.5 If you currently use an electronic address book or calendar, compare its features to those of a database system.

- **a)** Does it support physical or logical data independence?
- **b)** How does sharing work?
- **c)** Can you search for records? What is the search language?
- **d)** Is there a DBA? What are the responsibilities of the DBA?

1.6 If you know how to use a version control system (such as CVS or Subversion), compare its features to those of a database system.

- **a)** Does a version control system have a concept of a record?
- **b)** How does checkin/checkout correspond to database concurrency control?
- **c)** How does a user perform a commit? How does a user undo uncommitted changes?
- **d)** Many version control systems save updates in *difference files*, which are small files that describe how to transform the previous version of the file into the new one. If a user needs to see the current version of the file, the system starts with the original file and applies all of the difference files to it. How well does this implementation strategy satisfy the needs of a database system?

PROJECT-BASED EXERCISE

1.7 Investigate whether your school administration or company uses a database system. If so:

- **a)** What employees explicitly use the database system in their job? (As opposed to those employees who run "canned" programs that use the database without their knowledge.) What do they use it for?
- **b)** When a user needs to do something new with the data, does the user write his own query, or does someone else do it?
- **c)** Is there a DBA? What responsibilities does the DBA have? How much power does the DBA have?