DATABASE MANAGEMENT: A SYSTEMS APPROACH USING JAVA

Edward Sciore
Boston College

CONTENTS

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, etc. Figure 1-1 depicts a database that holds information about students in a university and the courses they have taken. This 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.
- 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.

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

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*, that 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 seem 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 of 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 shall 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.

```
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 3 TAB m a x ...
```

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

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, that enable fast searching. A database system needs to implement its files using analogous data structures. In fact, a record type might have 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 and see the 40 available seats, and then 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 then 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 undo-able 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 6 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 1,000 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 1,000 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 above 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.

```
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

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.

> *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 that lists the relevant information about the course taken.

Similarly, the faculty in the university my be interested in having a table 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*.

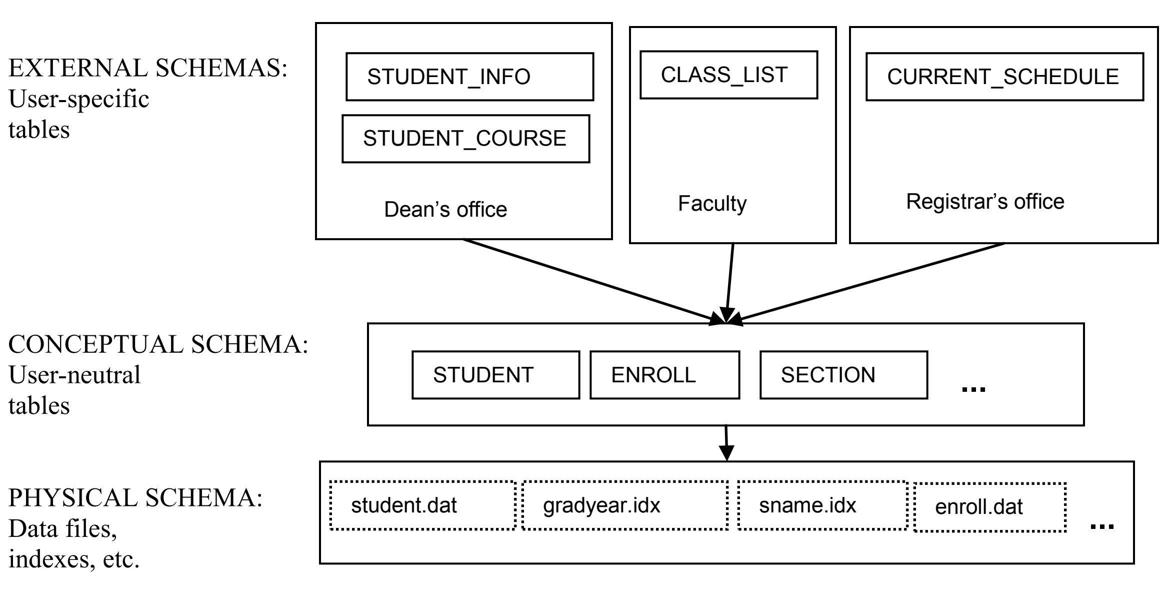


Figure 1-4: The three schema levels

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.*

Logical data independence provides three benefits:

- customized 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 *en.wikipedia.org/wiki/Database_management_system#History* 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) How does the version-control system resolve conflicting updates? Is this approach appropriate for a database system?
e) 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?

PART 1
Relational Databases

The relational database model is the simplest and currently the most prevalent means of organizing and accessing data. Most commercial products are either based entirely on the relational model (e.g. *Access* and *MySQL*), or are based on models that extend or generalize the relational model, such as object-oriented systems (e.g. *Objectivity* and *ObjectStore*), object-relational systems (e.g. *Oracle* and *PostgreSQL*), or semi-structured systems (e.g. *SODA2*). Regardless of the model, however, the basic relational concepts – *tables*, *relationships*, and *queries* – are fundamental, and these concepts form the core around which the various generalizations can occur.

Part 1 introduces the basic concepts underlying the relational model. Chapter 2 covers the structural aspects: how tables are specified and how they are related to each other. Chapter 3 looks at design issues, providing criteria for distinguishing good database designs from bad ones, and introducing techniques for creating good designs. Chapter 4 examines the principles behind relational data manipulation, and provides a basic introduction to SQL, the official standard query language. Chapter 5 considers the problems of ensuring that users see only the data they are authorized to see, and that they do not corrupt the data. And Chapter 6 examines two constructs – materialized views and indexes – that database systems use to improve the efficiency of queries.

2
DATA DEFINITION

This chapter examines how a relational database is structured and organized. We introduce tables, which are the building blocks of a relational database. We show how keys are used to identify records in a table, and how foreign keys are used to connect records from different tables. The importance of keys and foreign keys are tied to their use as constraints. We examine these and other constraints, such as null value constraints and integrity constraints. Finally, we see how tables and their constraints can be specified in SQL, the current standard database language.

2.1 Tables

The data in a relational database system is organized into *tables*. Each table contains 0 or more *records* (the rows of the table) and 1 or more *fields* (the columns of the table). Each record has a value for each field.

Each field of a table has a specified *type*, and all record values for that field must be of that type. The database of Figure 1-1 is fairly simple, in that all of the fields either have the type *integer* or *character string*. Commercial database systems typically support many types, including various numeric, string, and date/time types; these types are discussed in Section 4.3.2.[†]

In Chapter 1 we defined a relational schema to be the fields of each table and their types. Often, when discussing a database, it is convenient to ignore the type information; in such cases, we can write the schema by simply listing the field names for each table. Figure 2-1 gives such a schema for the university database.

```
STUDENT(SId, SName, GradYear, MajorId)
DEPT(DId, DName)
COURSE(CId, Title, DeptId)
SECTION(SectId, CourseId, Prof, YearOffered)
ENROLL(EId, StudentId, SectionId, Grade)
```

Figure 2-1: The schema of the university database

[†] There are also many types not covered in this book. One particularly funny type name is a string of bytes called a *BLOB*, which is short for “Binary Large OBject”. BLOBs are useful for holding values such as images.

2.2 Null Values

A user may not know what value to store in a record, because the value is either unknown, unavailable, or nonexistent. In such a case a special value, called a *null value*, is stored there.

A null value denotes a value that does not exist or is unknown.

Do not think of a null as a real value; instead, think of it as a placeholder that means "I have no clue what the value is". Nulls behave strangely in queries. Consider the STUDENT table of Figure 1-1, and suppose that the records for Joe and Amy both have a null value for *GradYear*. These nulls indicate that we do not know when (or if) they graduated. So if we ask for all students that graduated in 2004, neither student will appear in the output. If we ask for all students that didn't graduate in 2004, neither student will appear. If we count the records for each *GradYear*, they won't be counted. In fact, if the search condition of a query depends on the graduation year, then those two records will not contribute to the output, no matter what the query is. This behavior is entirely proper, because if we don't know someone's graduation year then we really can't say anything about them, either positive or negative. But even though the behavior is proper, it is nevertheless strange, and totally unlike that of any other value.

A weird aspect of nulls is that they are neither equal nor unequal to each other. Continuing the above example, suppose that we ask whether Joe and Amy have the same graduation year; the answer is "no", because we do not know if they do. Similarly, if we ask whether they have different graduation years, the answer is also "no" for the same reason. Even weirder is that the answer is "no" even if we ask whether Joe has the same graduation year as himself!

Because null values do not behave nicely, their presence in a table adds a semantic complexity that forces users to always think in terms of special cases. It would be ideal if tables never had nulls, but in practice nulls do occur.

Null values occur for two reasons:

- *Data collection may be incomplete.*
- *Data may arrive late.*

Data collection, no matter how conscientious, is sometimes incomplete. Sometimes, personal data (such as a student's home phone number) is withheld. Other times, data gets lost before it can be added to the database. For example, consider the SECTION table. When the university schedules a new section, all of the information about it is available, and so the corresponding record will not have nulls. But what about sections that were offered before the database was created? The information about those sections must be culled from external sources, such as old course catalogs. If those catalogs did not list the professor's name, then those records must be given a null value for the field *Prof*.

A field value may become available only after the record is created. For example, an ENROLL record might be created when the student registers for a section; the value for *Grade* will therefore start out null and will be filled in when the student completes the course. Similarly, a STUDENT record might be inserted when the student is admitted, but before the student's major is known; in that case the value for *MajorId* is initially null.

Note that these two scenarios produce null values having different meanings. A null value in the first scenario means "value not known", whereas a null value in the second scenario means "value doesn't exist". These different meanings can cause confusion. For example, suppose an ENROLL record has a null value for *Grade*; we cannot tell if the grade is missing or if the grade has not yet been given. One way to reduce this confusion is to avoid using "value doesn't exist" nulls; see Exercise 2.1.

The creator of a table specifies, for each field, whether that field is allowed to contain null values. In making this decision, the creator must balance the flexibility of allowing nulls with the added complexity they incur.

2.3 Keys

In the relational model, a user cannot reference a record by specifying its position (as in "I want the second record in STUDENT"). Instead, a user must reference a record by specifying field values (as in "I want the record for the student named Joe who graduated in 1997"). But not all field values are guaranteed to uniquely identify a record. For example, specifying the student name and graduation year (as in the above request) is not sufficient, because it is possible for two students to have the same name and graduation year. On the other hand, supplying the student ID would be sufficient, because every student has a different ID.

A unique identifier is called a *superkey*.

> *A superkey of a table is a field (or fields)*
> *whose values uniquely identify the table's records.*

Note that adding a field to a superkey always produces another superkey. For example, *SId* is a superkey of STUDENT, because no two records can have the same *SId* value. It follows trivially that {*SId*, *GradYear*} is also a superkey, because if two records cannot have the same *SId* values, they certainly cannot have the same *SId* and *GradYear* values.

A superkey with superfluous fields satisfies the definition but not the spirit of a unique identifier. That is, if I can identify a STUDENT record by giving its *SId* value, then it is misleading and inappropriate to also specify a *GradYear* value. We define a *key* to be a superkey without the superfluous fields.

> *A key is a superkey having the property that*
> *no subset of its fields is a superkey.*

How do we know when something is a key (or superkey)? We have already asserted that *SId* is a superkey of STUDENT, and that {*SName*, *GradYear*} is not. We justified those assertions by considering what records the table might reasonably contain. Note that this justification had nothing to do with the current contents of the table.

In fact, you cannot determine the possible keys of a table by looking at its contents. For example, consider the fields *SName* and *DName* in the STUDENT and DEPT tables of Figure 1-1. These columns do not contain duplicate values; does this mean that *SName* and *DName* could be keys of their respective tables? On one hand, the university controls the names of its departments, and will make sure that no two departments have the same name; thus *DName* could be a key. On the other hand, there is nothing to keep two students from having the same name; thus *SName* cannot be a key.

In other words, the *intent* of a table, not its content, determines what its keys are. A table's keys must be specified explicitly. Each specified key acts as a constraint that limits the table's contents. For example, specifying that *DName* is a key of DEPT constrains the table so that no two departments can have the same name.

Here are some additional examples of keys as constraints. If we say that {*StudentId*, *SectionId*} is a key for ENROLL, then we are asserting that there cannot be two records that have the same values for both fields; in other words, a student cannot enroll in the same section twice. If we consider the table SECTION, each of the following assumptions results in a different key:

- If a professor teaches at most one section a year, then {*Prof*, *YearOffered*} is a key.
- If a course can have at most one section per year, then {*CourseId*, *YearOffered*} is a key.
- If a professor teaches at most one section of a given course in a year, then {*CourseId*, *Prof*, *YearOffered*} is a key.

Most relational databases do not require a table to have a key. However, the absence of a key means that the table is allowed to contain duplicate records (see Exercise 2.4). Duplicate records can cause problems, because they cannot be distinguished from each other. (Remember, we can access a record only by its values, not by its position.) For example, suppose that there are two students having the same name, major, and graduation year. If we remove the key field *SId* from the STUDENT table, then the records for these two students are duplicates. There now is no way to treat these students independently. In particular, we cannot change the major of one but not the other, nor can we enroll them in different courses or give them different grades.

Although a table can have several keys, one key is chosen to be the *primary key*. As we shall see, records are referenced by their primary key; thus in a well-designed database, each primary key should be as natural and as easy to understand as possible. Consider for example the third bullet point above. Even if {*CourseId*, *Prof*, *YearOffered*} were a key of SECTION, it probably would be unwise to choose it as the primary key, since the field *SectId* is much more intuitive.

ID numbers are often used as primary keys, because they are simple and intuitive. The keys of our example database are all ID numbers. These keys have just one liability, which is that their values are artificial. Students and courses don't have ID numbers in the real world; these numbers exist only inside the database. As a consequence, a new ID value has to be generated each time a new record is created. In some situations, the user will take responsibility for generating the new IDs. (For example, the university may have a specific algorithm for generating student IDs.) In other cases the user doesn't care, and allows the database system to generate the ID value automatically.

When choosing a primary key, it is essential that its values can never be null. Otherwise it would be impossible to ensure that keys are unique, because there would be no way to tell if the record has the same key as another record.

2.4 Foreign Keys and Referential Integrity

The information in a database is split among its tables. However, these tables are not isolated from each other; instead, a record in one table may contain values relevant to a record in another table. This correspondence is achieved via *foreign keys*.

> *A foreign key is a field (or fields) of one table*
> *that corresponds to the primary key of another table.*

A foreign key value in one record uniquely identifies a record in the other table, thereby creating a strong logical connection between the two records. Consider Amy's STUDENT record, which has a *MajorId* value of 20. If we assume that *MajorId* is a foreign key of DEPT, then there is a connection between Amy's record and the record for the math department. In other words, Amy is a math major.

The connection between keys and their foreign keys works in both directions. Given a STUDENT record we can use its foreign key value to determine the DEPT record corresponding to the student's major, and given a DEPT record we can search the STUDENT table to find the records of those students having that major.

As with keys, the existence of a foreign key cannot be inferred from the database. For example when we look at Figure 1-1, it sure seems like *MajorId* is a foreign key of DEPT. But this could be an illusion. Perhaps majors are interdisciplinary and not associated with departments, and the values in *MajorId* refer to a list of major codes printed in the course catalog but not stored in the database. In this case, there is no connection between *MajorId* and the rest of the database, and the database can say nothing about Amy's major other than "she has major #20".

The foreign keys for a table must be specified explicitly by the table's creator. We shall assume that five foreign keys have been specified for the database of Figure 1-1. These keys are listed in Figure 2-2.

MajorId in STUDENT is a foreign key of DEPT;
DeptId in COURSE is a foreign key of DEPT;
CourseId in SECTION is a foreign key of COURSE;
StudentId in ENROLL is a foreign key of STUDENT;
SectionId in ENROLL is a foreign key of SECTION.

Figure 2-2: Foreign keys for the university database

Why bother to specify a foreign key? What difference does it make? To some extent, it makes no difference. Even if no foreign keys are specified, a user can still write queries that compare would-be foreign keys to their associated keys. There are two reasons why foreign keys are important.

Specifying a foreign key has two benefits:

- *It documents the connection between the two tables.*
- *It allows the database system to enforce referential integrity.*

Consider the field *MajorId*. If this field is not specified as a foreign key, then a user is forced to guess at its meaning. By specifying the field as a foreign key, the creator documents its purpose – namely, as a reference to a particular DEPT record. This documentation helps users understand how the tables are connected to each other, and tells them when it is appropriate to connect values between tables.

A foreign key has an associated constraint known as *referential integrity*, which limits the allowable values of the foreign key fields. Specifying a foreign key enables the database system to enforce this constraint.

Referential integrity requires each non-null foreign key value to be the key value of some record.

Consider Figure 1-1. The STUDENT records satisfy referential integrity, because their *MajorId* values are all key values of records in DEPT. Similarly, the ENROLL records satisfy referential integrity because each *StudentId* value is the *SId* of some student, and each *SectionId* value is the *SectId* of some section.

Note that the definition applies only to non-null foreign keys. As usual, null values are a special case. A foreign key can be null and not violate referential integrity.

The database system checks for referential integrity whenever a user attempts to update the database. If the system determines that referential integrity would be violated, then the user's update request is rejected.

In order to understand referential integrity better, let's look more closely at the foreign key *MajorId*, which connects STUDENT records to their major DEPT record. Figure 2-3 depicts the math-department record and the records of two math majors. Each arrow in the figure indicates the correspondence between the foreign key value and its associated key value.

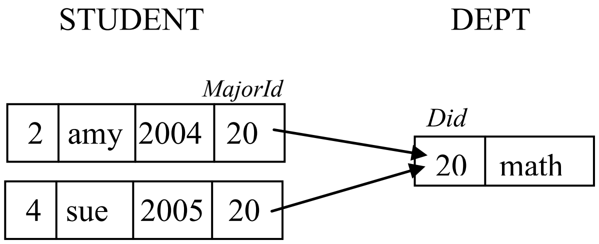


Figure 2-3: An example of referential integrity

Using Figure 2-3 as an example, there are four actions that would violate referential integrity:

- Modifying the *DId* value of the math-department record would cause the records for Amy and Sue to have invalid *MajorId* values.
- Deleting the math-department record would have the same effect.
- Modifying the *MajorId* value for Amy to a nonexistent department ID (such as 70) would cause her record to no longer reference a valid DEPT record.
- Inserting a new STUDENT record having a *MajorId* value of 70 would have the same effect.

There are two relevant actions that are not on the above list: deleting one of the STUDENT records, and inserting a new DEPT record. Exercise 2.5 asks you to show that these actions cannot possibly violate referential integrity.

Of the above actions, the first two (which update DEPT) are very different from the last two (which update STUDENT). In the last two cases, the requested action is meaningless; it makes absolutely no sense to have a STUDENT record whose foreign key value does not correspond to an existing key. In the first two cases, the requested action is meaningful; the problem is that modifying (or deleting) the math-department record affects the STUDENT records that refer to it. Before performing that action, the user is therefore obligated to do something about these other records. Let's look at the kinds of things a user could do.

First suppose that the user wants to delete the math-department record. There are two ways to deal with the existing math majors. One way is to modify their foreign key value, either by assigning them to a new department (say, department 10), or by giving them a null major. The other way is to just delete the math majors from the STUDENT table. In this particular example the first option is more appropriate, since the second option seems overly harsh to the students.

For a case where the second option is more appropriate, consider the foreign key *StudentId* of ENROLL. Here if we delete a STUDENT record, then it makes sense to delete the enrollment records corresponding to that student. This option is called *cascade delete*, because the deletion of one record can delete others; and if those other deleted records are referred to by other still other records, a chain of deletions can ensue.

Now suppose that the user wants to modify the math-department record, say by changing its ID from 20 to 70. The user has two reasonable ways to deal with the existing math majors: their *MajorId* values can be set to the new key value, or to null. Although both options are possible, the first one is almost always preferable. This first option is called *cascade update*, because the update to DEPT causes a corresponding update to STUDENT.

Although a user deleting (or modifying) a record is obligated to deal with records having foreign-key references to it, this obligation can often be handled automatically by the database system. When the foreign key is specified, the table creator can also specify whether one of the above options should occur. For example, the specification of foreign key *MajorId* might assert that deletions to DEPT records should be handled by setting the major of affected STUDENT records to null, whereas the specification of foreign key *StudentId* might assert that deletions to STUDENT records should be handled by also deleting any affected ENROLL records.

2.5 Integrity Constraints

A database typically has several *constraints* specified on it.

> *A constraint describes the allowable states*
> *that the tables in the database may be in.*

The database system is responsible for ensuring that the constraints are satisfied. Typically, it does so by refusing to execute any update request that would violate a constraint.

In this chapter we have so far seen three different kinds of constraint:

- *Null value constraints* specify that a particular field must not contain nulls.
- *Key constraints* specify that two records cannot have the same values for the key’s fields.
- *Referential integrity constraints* specify that a foreign key value of one record must be the key value of another record.

Another kind of constraint is called an *integrity constraint*.

> *Integrity constraints specify what it means*
> *for the database to reflect reality.*

An integrity constraint may impose requirements on a single record, such as that the values of a field must be in a specified range (e.g. at Boston College, *GradYear* is at least 1863) or have specified values (e.g. *Grade* must be either 'A', 'A-', etc.). An integrity constraint may also impose a requirement on a table as a whole, such as that the ENROLL table cannot have more than 20 records per section, or that the SECTION table cannot have a professor teaching more than 2 sections per year. An integrity constraint may even involve multiple tables, such as the requirement that a student cannot enroll in a course more than once.

> *Integrity constraints have two purposes:*
> - *They can detect bad data entry.*
> - *They can enforce the "rules" of the organization.*

A constraint to detect bad data entry would be "*GradYear* values must be at least 1863". For example, this constraint will detect a typo value of "1009" instead of "2009". Unfortunately, this constraint has limited usefulness, because while it can detect the obviously bad data, it will miss the subtly bad. For example, the constraint cannot detect the typo value of "2000" instead of "2009".

The more important purpose of integrity constraints is to enforce the "rules" of the organization. For example, many universities have rules such as "students cannot enroll in a course without having taken all of its prerequisites" or "students cannot enroll in a course that they have already passed". Checking for these constraints can keep students from doing something they will regret later.

The decision to specify an integrity constraint is a two-edged sword. On one hand, the enforcement of a constraint helps to weed out bad data. On the other hand, it takes time (and possibly a lot of time) to enforce each constraint. As a practical matter, the database designer must weigh the time required to enforce a constraint against its expected benefits. For example, computer science majors did not exist prior to 1960, so a valid constraint is that students graduating before 1960 should not have "compsci" as their major. Should we specify this constraint in the database?

The answer depends on how the database is being used. If the database is being used by the admissions office, then new STUDENT records correspond to entering students, none of which will have a graduation year prior to 1960; thus the constraint is unlikely to be violated, and checking it would be a big waste of time. However, suppose that the database is being used by the alumni office, where they have a web site that allows alumni to create their own STUDENT records. In this case the constraint could be of use, by detecting inaccurate data entry.

2.6 Specifying Tables in SQL

SQL (pronounced "ess-kew-ell") is the official standard relational database language. The language is *very* large. In this book we focus on the most important aspect of SQL; Sections 2.8 and 4.7 have references to more comprehensive SQL resources.

You specify a table in SQL using the *create table* command. This command specifies the name of the table, the name and type of its fields, and any constraints. For example, Figure 2-4 contains SQL code to create the STUDENT table of Figure 1-1.

```
create table STUDENT (
   SId int not null,
   SName varchar(10) not null,
   MajorId  int,
   GradYear int,

   primary key (SId),
   foreign key (MajorId) references DEPT
      on update cascade
      on delete set null,
   check (SId > 0),
   check (GradYear >= 1863)
)
```

Figure 2-4: The SQL specification of the STUDENT table

SQL is a free-form language – newlines and indentation are all irrelevant. Figure 2-4 is nicely indented in order to make it easier for people to read, but the database system doesn't care. SQL is also case-insensitive – the database system does not care whether keywords, table names, and field names are in lower case, upper case, or any combination of the two. Figure 2-4 uses capitalization solely as a way to enhance readability.

The field and constraint information for a table is written as a comma-separated list within parentheses. There is one item in the list for each field or constraint. Figure 2-4 specifies four fields and six constraints.

A field specification contains the name of the field, its type, and optionally the phrase "not null" to indicate a null-value constraint. SQL has numerous built-in types, but for the moment we are interested in only two of them: integers (denoted by the keyword *int*) and character strings (denoted by *varchar(n)* for some n). The keyword *varchar* stands for "variable length character string", and the value *n* denotes the maximum length of the string. For example, Figure 2-4 specifies that student names can be anywhere from 0 to 10 characters long. Chapter 4 will discuss SQL types in more detail.

Figure 2-4 also illustrates primary key, foreign key, and integrity constraints. In the primary key specification, the key fields are placed within parentheses, and separated by commas. The foreign key specification also places the foreign key fields within parentheses, and specifies the table that the foreign key refers to. That constraint also specifies the action that the database system is to perform when a referenced record (i.e., a record from DEPT) is deleted or modified. The phrase "on update cascade" asserts that whenever the key value of a DEPT record is modified, the *MajorId* values in corresponding STUDENT records will likewise be modified. The phrase "on delete set null" asserts that whenever a DEPT record is deleted, the *MajorId* values in corresponding STUDENT records will be set to null.

The action specified with the *on delete* and *on update* keywords can be one of the following:

- *cascade*, which causes the same query (i.e. a delete or update) to apply to each foreign-key record;
- *set null*, which causes the foreign-key values to be set to null;
- *set default*, or which causes the foreign-key values to be set to their default value;
- *no action*, which causes the query to be rejected if there exists an affected foreign-key record.

Integrity constraints on individual records are specified by means of the *check* keyword. Following the keyword is a Boolean expression that expresses the condition that must be satisfied. The database system will evaluate the expression each time a record changes, and reject the change if the expression evaluates to *false*. Figure 2-4 specifies two such integrity constraints.

To specify an integrity constraint that applies to the entire table (or to multiple tables), you must create a separate *assertion*. Because assertions are specified in terms of queries, we shall postpone their discussion until Chapter 5.

2.7 Chapter Summary

- The data in a relational database is organized into *tables*. Each table contains 0 or more *records* and 1 or more *fields*.

- Each field has a specified *type*. Commercial database systems support many types, including various numeric, string, and date/time types.

- A *constraint* restricts the allowable records in a table. The database system ensures that the constraints are satisfied by refusing to execute any update request that would violate them.

- There are four important kinds of constraint: *integrity constraints, null value constraints, key constraints,* and *referential integrity constraints*. Constraints need to be specified explicitly; they cannot be inferred from the current contents of a table.

- An *integrity constraint* encodes "business rules" about the organization. An integrity constraint may apply to an individual record ("a student's graduation year is at least 1863"), a table ("a professor teaches at most two sections per year"), or database ("a student cannot take a course more than once").

- A *null value* is a placeholder for a missing or unknown value. A *null value constraint* asserts that a record cannot have a null value for the specified field. Null values have a complex semantics, and should be disallowed for as many fields as possible. However, null values cannot always be avoided, because data collection may be incomplete and data may arrive late.

- A *key* is a minimal set of fields that can be used to identify a record. Specifying a key constrains the table so that no two records can have the same values for those fields. Although a table can have several keys, one key is chosen to be the *primary key*. Primary key fields must never be null.

- A *foreign key* is a set of fields from one table that corresponds to the primary key of another table. A foreign key value in one record uniquely identifies a record in the other table, thereby connecting the two records. The specification of a foreign key asserts *referential integrity*, which requires each non-null foreign key value to be the key value of some record.

- A table is specified in SQL using the *create table* command. This command specifies the name of the table, the name and type of its fields, and any constraints.
    - An integrity constraint on individual records is specified by means of the *check* keyword, followed by the constraint expression.
    - A null value constraint is specified by adding the phrase “not null” to a field specification.
    - A primary key is specified by means of the *key* keyword, followed by placing the key fields within parentheses, separated by commas.
    - A foreign key is specified by means of the *foreign key* keywords, followed by placing the foreign key fields are placed within parentheses, and then specifying the table that the foreign key refers to.

- A foreign key constraint can also specify the action that the database system is to perform when an update does violate referential integrity. The action can be one of *cascade*, *set null*, *set default*, and *no action*.

2.8 Suggested Reading

The inspiration behind relational database systems came from mathematics, in particular the topics of relations (i.e. multi-valued functions) and set theory. Consequently, tables and records are sometimes called “relations” and “tuples”, in order to emphasize their mathematical nature. We have avoided that terminology in this book, because “table” and “record” have a more intuitive feel. A more formal, mathematically-based presentation of the relational model appears in [Date and Darwen, 2006].

2.9 Exercises

CONCEPTUAL EXERCISES

2.1 Section 2.2 mentions that that ENROLL records should have a null value for *Grade* while the course is in progress, and that STUDENT records should have a null value for *MajorId* until a student declares a major. However, this use of null values is undesirable. Come up with a better way to represent in-progress courses and undeclared majors.

2.2 Explain what it would mean if the field *{MajorId, GradYear}* were a key of STUDENT. What would it mean if {*SName*, *MajorId*} were a key of STUDENT?

2.3 For each of the tables in Figure 1-1, give a reasonable key other than its ID field. If there is no other reasonable key, explain why.

2.4 Show that if a table has at least one key, then the table can never contain duplicate records. Conversely, show that if a table does not have a key, then duplicate records are possible.

2.5 Figure 2-2 asserts that *MajorId* is a foreign key of DEPT.
a) Explain why deleting a record from STUDENT does not violate referential integrity.
b) Explain why inserting a record into DEPT does not violate referential integrity.

2.6 Give the SQL code to create the other tables in Figure 1-1. Specify some reasonable integrity constraints on the individual records.

2.7 An online bookseller stores information in the following tables:

BOOK(<u>BookId</u>, Title, AuthorName, Price)
CUSTOMER(<u>CustId</u>, Name, Address)
CART_ITEM(CustId, BookId)
PURCHASE(<u>PId</u>, PurchaseDate, Cust#)
PURCHASED_ITEM(PId, BookId)

The specified keys are underlined; the tables PURCHASED_ITEM and CART have no specified key. Every customer has a shopping cart, which contains books that the customer has selected but not paid for; these books are listed in the CART table. At any time, a customer can purchase the books in his cart. When this occurs, the following things happen: a new record is created in PURCHASE; each book in the cart is added to the PURCHASED_ITEM table; and the customer's cart is emptied.

a) What should we do about keys for PURCHASED_ITEM and CART?
b) Give the foreign keys of each table.
c) List in English some reasonable integrity constraints that these tables might have.
d) Specify these tables in SQL.

2.8 Explain what the difference would be if we changed the foreign key specification in Figure 2-4 to the following:

```
on update no action
on delete cascade
```

3

DATA DESIGN

Designing a database is an important task, and is much more difficult than it might appear. This chapter presents a methodology for designing relational databases. In this methodology, we begin by analyzing the data and structuring it into *entities* and *relationships*. We use a *class diagram* to express this structure.

A class diagram eventually needs to be turned into a relational schema. We will examine an algorithm that performs this transformation. This algorithm can handle only certain class diagrams (namely, those diagrams where the relationships correspond to foreign keys). We therefore consider what to do about other, more general relationships between classes. We show that the presence of such relationships indicates an incomplete design; we also show how to restructure the class diagram so that it not only is better, but it also no longer needs the general relationships.

We also address the question of what it means to be a good design. We examine two kinds of problematic relationships – *inadequate relationships* and *redundant relationships* – and show how to eliminate them from a class diagram. We then introduce *functional dependencies*, and show how they can be used as a tool to identify and fix redundancies in a class diagram.

3.1 Designing Tables is Difficult

A relational database is just a set of tables. If you want to create a new database, all you have to do is figure out what data you need to keep, and design some tables to hold that data. What is so difficult about that?

Let's try a simple example. Suppose that you want to use a database to catalog your CD music collection. Each CD has some data you want to store, such as its title, release year, band, and genre. Each individual track on the CD also has data, such as its track number, song title, and duration. So you create two tables: one to hold CD data, and one to hold track data. The tables have the following schema.

```
CD(CDId, Title, ReleaseYear, Band, Genre)
TRACK(TrackId, CDId, Track#, SongTitle, Duration)
```

You decide to create artificial fields *CDId* and *TrackId* to be the key of each table[†]. These fields are underlined in the above schema. You also specify that the field *CDId* inTRACK is a foreign key of CD; this foreign key is italicized in the schema.

Frankly, there is nothing wrong with this design, provided that all of your CDs fit this simple structure and that you aren't interested in saving additional data. But what would you do about the following issues?

- A "best of" CD contains tracks from different CDs. Should the records in your TRACK table reference the original CD? What if you don't own the original CD?
- What if you burn your own CD that contains the same songs as an existing CD but in a different order? Is it the same CD? What is its title?
- Some CDs, such as "Best of 90's Salsa" contain tracks from multiple bands. Do you have to store the band name with each track individually?
- Suppose you burn a CD that contains music from different genres. Do you save the genre with the track, or do you create a new genre for the CD, such as "party mix"?
- Do you want to keep data on the individual performers in the band? If so, do you want to be able to know about band members who create solo albums or move to different bands? A track may have appearances by "guest artists" who perform only on that track. How will you represent this?
- Do you want to keep data on each song, such as who wrote it and when? Often a track will be a cover of a song that was written and performed by another band. How will you represent that? Do you want to be able to create a playlist of every version of a song in the order of its performance?
- Do you want to save song lyrics? What about the author of those lyrics? What will you do with songs that do not have lyrics?

We could go on, but the point is this: As you widen the scope of what you want the database to contain, complexity tends to creep in. Choices need to be made about how specific the data will be (e.g. should *genre* be stored with a band, a CD, a track of a CD, or a song?), how extensive the data will be (e.g. do we really want to know who wrote a song?), and how inter-related the data will be (e.g. should we be able to compare different versions of the same song?).

Try for yourself to come up with a good set of tables to hold all of this data. How do you decide what tables to create, and what fields those tables should have? What should the foreign keys be? And most importantly, how do you *know* that your design is a good one? What started out as a simple, easily-understood, two-table database has turned into something much less simple and quite likely much less easy to understand.

Many corporate databases are even more complex. For example, peruse the *amazon.com* or *imdb.com* websites and note the different kinds of data they manage. A hit-or-miss

[†] A track ID uniquely identifies a track of any CD, as opposed to a track#, which is the location of the track on a particular CD.

design strategy just isn't going to work for these kinds of schemas. We need a methodology that can guide us through the design process.

The design of a database has consequences that can last far into the future. As the database is used, new data is collected and stored in it. This data can become the lifeblood of the organization using the database. If it turns out that the database doesn't contain the right data, the organization will suffer. And because missing data often cannot be re-collected, it can be very difficult to restructure an existing database. An organization has a lot of incentive to get it right the first time.

3.2 Class Diagrams

Over the years, people have discovered that the best way to design a relational database is to *not* think about tables. The reason is that a database is a model of some portion of the world, and the world is just not organized as tables. Instead, the world tends to be organized as a network of related entities. Thus the best approach is to first construct a *class diagram*, which is a model of the world in terms of entities and relationships, and to then translate that diagram to a relational schema. This section introduces class diagrams and their terminology. The following sections show how to translate a class diagram to a relational schema, and how to create a well-designed diagram.

3.2.1 Classes, relationships, and attributes

People have developed many different notational systems for depicting entities and their relationships. The pictures we shall draw are called *class diagrams*, and are taken from the design language *UML*.

> *A class diagram describes the entities that the database should hold and how they should relate to each other.*

A class diagram uses three constructs: *class*, *relationship*, and *attribute*. A class denotes a type of entity, and is represented in the diagram by a box. A relationship denotes a meaningful correspondence between the entities of two or more classes, and is represented by a line connecting those classes. An attribute is used to describe an entity. Each class will have zero or more attributes, which are written inside its box. An entity will have a value for each attribute of its class.

Figure 3-1 depicts the class diagram for our example university database, with the addition of a class PERMIT that holds information about parking permits. In this diagram, the class STUDENT participates in three relationships; the classes DEPT, ENROLL, SECTION, and COURSE participate in two relationships each; and the class PERMIT participates in one relationship.

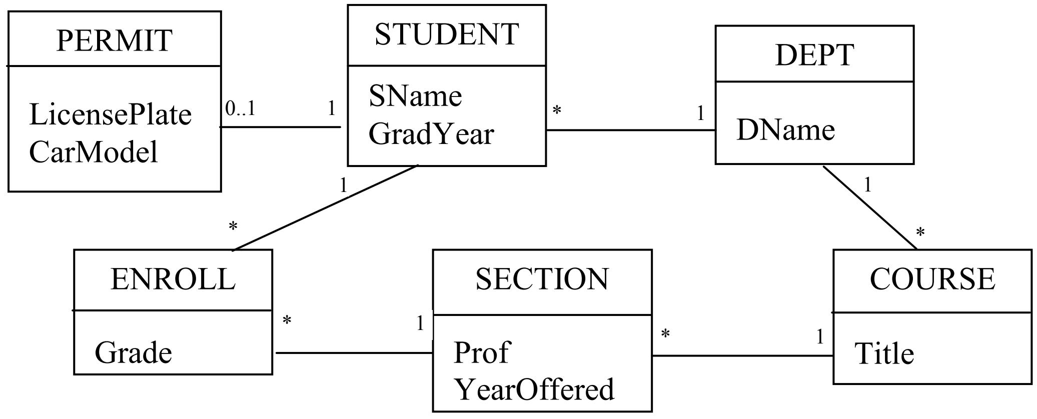


Figure 3-1: A class diagram for the university database

Most concepts in a class diagram have a corresponding concept in the relational model. A class corresponds to a table, and its attributes correspond to the fields of the table. An entity corresponds to a record. A relationship is the one concept that does not have a direct analog in the relational model, although a foreign key comes close. These correspondences will form the core of the algorithm that translates class diagrams to relational schemas, as we shall see in Section 3.3.

3.2.2 Relationship cardinality

Each side of a relationship has a *cardinality* annotation.

The cardinality annotations indicate, for a given entity, how many other entities can be related to it.

Consider the relationship between STUDENT and DEPT in Figure 3-1. The "*" next to STUDENT means that a department can have any number of (including zero) student majors, and the "1" next to DEPT means that each STUDENT record must have exactly one associated major department. This relationship is called a *many-one* relationship, because a department can have many students, but a student can have only one department. Many-one relationships are the most common form of relationship in a database. In fact, all but one of the relationships in Figure 3-1 are many-one.

Now suppose that we changed the relationship between STUDENT and DEPT so that the annotation next to DEPT was a "*" instead of a "1". This annotation would denote that a student could declare several (including zero) major departments, and each department could have several student majors. Such a relationship is called a *many-many* relationship.

The relationship between STUDENT and PERMIT is an example of a *one-one* relationship, meaning that each student can have one parking permit, and each permit is for a single student. The relationship is not exactly symmetrical, because a permit must have exactly one related student, whereas a student can have at most one (and possibly zero) related permits. We denote this latter condition by the "0..1" annotation next to PERMIT in the class diagram.

In general, UML allows annotations to be of the form "N..M", which denotes that a record must have at least N and at most M related records. The annotation "1" is a shorthand for "1..1", and "*" is a shorthand for "0..*". The annotations "1", "*", and "0..1" are the most common in practice, and are also the most important for database design. We shall consider them exclusively in this chapter.

3.2.3 Relationship strength

Each side of a relationship has a *strength*, which is determined by its annotation. The "1" annotation is called a *strong annotation*, because it requires the existence of a related entity. Conversely, the "0..1" and "*" annotations are called *weak annotations*, because there is no such requirement. For example in Figure 3.1, the relationship from STUDENT to DEPT is strong, because a STUDENT entity cannot exist without a related DEPT entity. On the other hand, the relationship from DEPT to STUDENT is weak, because a DEPT entity can exist without any related STUDENT entities.

We can classify a relationship according to the strength of its two sides. For example, all of the relationships in Figure 3-1 are *weak-strong*, because they have one weak side and one strong side. All many-many relationships are *weak-weak*, as are one-one relationships that are annotated by "0..1" on both sides. And a relationship that has "1" on both sides is called *strong-strong*.

3.2.4 Reading a class diagram

Because of its visual nature, a class diagram provides an overall picture of what data is available in the database. For example, by looking at Figure 3-1 we can see that *students* are *enrolled* in *sections* of *courses*. We can therefore assume that it should be possible to find, for example, all courses taken by a given student, all students taught by a given professor in a given year, and the average section size of each math course. We can also assume from the diagram that it will *not* be possible to determine which students used to be math majors (because only the current major is stored), or which professor is the advisor of a given student (because there is no advisor-advisee relationship).

The purpose of a class diagram is to express the design of the database as clearly as possible. Anything that the designer can do to improve the readability of the diagram is a good thing. For example, sometimes the meaning of a relationship is obvious by looking at the diagram, and sometimes it is not. In Figure 3-1, the relationship between STUDENT and ENROLL is obvious, as it denotes a student enrollment. However, the relationship between STUDENT and DEPT is less obvious. In such cases, designers often annotate the relationship by writing a description (such as "major") on the relationship's line in the class diagram.

3.3 Transforming Class Diagrams to Tables

3.3.1 Relationships and foreign keys

A class diagram consists of classes, attributes, and relationships. Classes correspond to tables, and attributes correspond to fields. But what do relationships correspond to?

The insight is this:

A foreign key corresponds to a weak-strong relationship.

For example, consider again the schema of the CD database that appeared at the beginning of this chapter:

```
CD(CDId, Title, ReleaseYear, Band, Genre)
TRACK(TrackId, CDId, Track#, SongTitle, Duration)
LYRICS(TrackId, Lyrics, Author)
```

Here, we have added a third table LYRICS, which holds the lyrics and author of each track that has lyrics. We use the field *TrackId* as the key of LYRICS, because each track can have at most one set of lyrics.

The field *CDId* in TRACK is a foreign key of CD, and the field *TrackId* in LYRICS is a foreign key of TRACK. The referential integrity constraints associated with these foreign keys assert that every TRACK record must have exactly one corresponding CD record, and every LYRICS record must have exactly one corresponding TRACK record. However, referential integrity says nothing about the other side of those relationships – A CD record can have any number of TRACK records that refer to it (even zero), and a TRACK record can have any number of LYRICS records.

In other words, the foreign key expresses a many-one relationship between CD and TRACK, and a many-one relationship between TRACK and LYRICS. (We can infer that the relationship between TRACK and LYRICS is actually one-one because the foreign key field is also the key.) We can depict this relational schema as the class diagram of Figure 3-2.

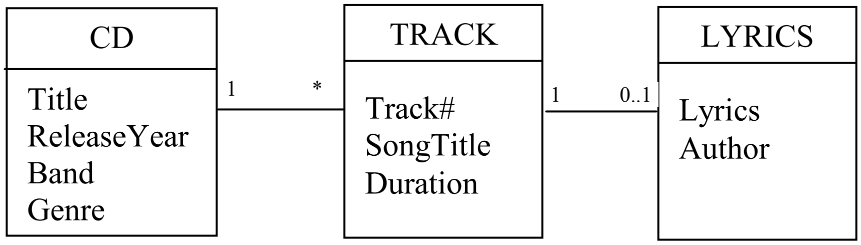


Figure 3-2: A class diagram for the example CD database

Note that the attributes of these classes are not identical to the fields of their corresponding tables. In particular, the key and foreign key fields are missing. In general, a class diagram should not have such fields. Foreign keys are not needed because their sole purpose is to implement a relationship, and relationships are already expressed directly in the diagram. And keys are not necessary in a class diagram because there are no foreign keys to refer to them.

*The attributes of a class express data values relevant to that class.
The class should not have attributes whose sole purpose
is to form a key or be a foreign key.*

It is perfectly legitimate (and not uncommon) for a class to have no attributes.

3.3.2 Transforming a relational schema

We can transform a relational schema to a class diagram by making use of the correspondence between foreign keys and relationships. The algorithm is straightforward, and appears in Figure 3-3.

1. Create a class for each table in the schema.

2. If table T_1 contains a foreign key of table T_2, then create a relationship between classes T_1 and T_2. The annotation next to T_2 will be "1". The annotation next to T_1 will be "0..1" if the foreign key is also a key; otherwise, the annotation will be "*".

3. Add attributes to each class. The attributes should include all fields of its table, except for the foreign keys and any artificial key fields.

Figure 3-3: The algorithm to transform a relational schema to a class diagram

We just saw one example of this algorithm, where we transformed the 3-table CD schema to the class diagram of Figure 3-2. For another example, consider the relational schema for the university database given in Figure 2-1, with the foreign key specifications of Figure 2-2. Applying this algorithm results in the class diagram of Figure 3-1 (minus the class PERMIT).

Although this algorithm is simple and straightforward, it turns out to be quite useful. The reason is that class diagrams tend to be easier to understand than relational schemas. For example, compare the relational schema of Figure 2-1 to the class diagram of Figure 3-1. The class diagram represents relationships visually, whereas the tables represent relationships via foreign keys. Most users find it easier to understand a picture than a linear list of table schemas.

Class diagrams also tend to be easier to revise than relational schemas. People have discovered that the best way to revise a relational schema is to transform its tables to a class diagram, revise the diagram, and then transform the diagram back to tables.

Knowing the class diagram for a relational schema also makes it easier to write queries. As we shall see in Chapter 4, often the hard part of writing a query is determining which tables are involved and how they get joined. The visual nature of a class diagram and its explicit relationships helps tremendously. In fact, if you showed the relational schema of Figure 2-1 to an experienced database user, that user would very likely translate these tables back into a class diagram, just to be able picture how the tables are connected. And this is but a simple example. In a more realistic database, class diagrams become indispensable.

3.3.3 Transforming a class diagram

Figure 3-4 presents an algorithm to transform a class diagram into a relational schema. This algorithm works by creating a foreign key in the schema for each relationship in the diagram.

1. Create a table for each class, whose fields are the attributes of that class.

2. Choose a primary key for each table. If there is no natural key, then add a field to the table to serve as an artificial key.

3. For each weak-strong relationship, add a foreign key field to its weak-side table to correspond to the key of the strong-side table.

Figure 3-4: The algorithm to transform a class diagram to a relational schema

For an example of this transformation, consider the class diagram of Figure 3-1. Suppose for simplicity that we choose to use an artificial key for each table. Then the resulting database would have the schema of Figure 3-5(a) (where primary keys are underlined and foreign keys are in italics).

```
PERMIT(PermitId, LicensePlate, CarModel, StudentId)
STUDENT(SId, SName, GradYear, MajorId)
DEPT(DId, DName)
ENROLL(EId, Grade, StudentId, SectionId)
SECTION(SectId, YearOffered, Prof, CourseId)
COURSE(Cid, Title, DeptId)
```

(a)

```
PERMIT(StudentId, LicensePlate, CarModel)
STUDENT(SId, SName, GradYear, MajorName)
DEPT(DName)
ENROLL(StudentId, SectionId, Grade)
SECTION(SectId, YearOffered, Prof, Title)
COURSE(Title, DeptName)
```

(b)

Figure 3-5: Two relational schemas generated from Figure 3-1

A variation of this algorithm is to intersperse the choice of primary keys and foreign keys, so that a foreign key field for a table could also be part of its primary key. The schema of Figure 3-5(b) uses this variation to generate as many non-artificial keys as possible. For example, the foreign key fields {*StudentId*, *SectionId*} are used as the primary key for ENROLL. This compound key is a good choice here because ENROLL does not have other tables referencing it, and so a corresponding multi-field foreign key will not get generated. In addition, the foreign key *StudentId* in PERMIT can be used as its primary key, because of the one-one relationship to STUDENT.

The algorithm of Figure 3-4 is only able to transform weak-strong relationships. The designer therefore has the responsibility to ensure that the class diagram has no weak-weak or strong-strong relationships. This issue will be addressed in Section 3.4.5.

3.4 The Design Process

Now that we understand how to transform a class diagram into a relational schema, we can turn to the most critical part of all – how to create a well-designed class diagram. In this section we shall introduce a design process that has six steps.

Six Steps in the Design of a Class Diagram

1. *Create a requirements specification.*
2. *Create a preliminary class diagram from the nouns and verbs of the specification.*
3. *Check for inadequate relationships in the diagram.*
4. *Remove redundant relationships from the diagram.*
5. *Revise weak-weak and strong-strong relationships.*
6. *Identify the attributes for each class.*

The following subsections will discuss these steps in detail. As an example, we shall apply our design process to the university database. We shall see which design decisions led to the class diagram of Figure 3-1, and examine other design decisions that would have resulted in other class diagrams.

We should point out that this design process (like most design processes) is flexible. For example, steps 3 through 6 need not be applied in a strict order. Moreover, the activity of one step may produce an insight that causes the designer to revisit previous steps. It is not uncommon for a designer to repeat the design process several times, with each iteration resulting in a more accurate and more detailed diagram.

3.4.1 Requirements analysis

The first step in the design of a database is to determine the data that it should hold. This step is called *requirements analysis*. There are several ways to obtain this information:

- you can ask the potential users of the database how they expect to use it;
- you can examine the data-entry forms that people will use;
- you can determine the ways that people intend to query the database;
- you can examine the various reports that will get generated from the data in the database.

The end product of this analysis is a text document that summarizes the requirements of the database; this document is called the *requirements specification*.

In our example, we assume that the university has asked us to design a database to support their student enrollment data. The result of our requirements analysis is the requirements specification of Figure 3-6.

The university is composed of departments. Academic departments (such as the Mathematics and Drama departments) are responsible for offering courses. Non-academic departments (such as the Admissions and Dining Hall departments) are responsible for the other tasks that keep the university running.

Each student in the university has a graduation year, and majors in a particular department. Each year, the students who have not yet graduated enroll in zero or more courses. A course may not be offered in a given year; but if it is offered, it can have one or more sections, each of which is taught by a professor. A student enrolls in a particular section of each desired course.

Each student is allowed to have one car on campus. In order to park on campus, the student must request a parking permit from the Campus Security department. To avoid misuse, a parking permit lists the license plate and model of the car.

The database should:

- allow students to declare and change their major department;
- keep track of parking permits;
- allow departments, at the beginning of each year, to specify how many sections of each course it will offer for that year, and who will teach each section;
- allow current students to enroll in sections each year;
- allow professors to assign grades to students in their sections.

Figure 3-6: The requirements specification for the university database

3.4.2 The preliminary class diagram

The second step in database design is to create a preliminary class diagram, by extracting relevant concepts from the requirements specification. We are primarily interested in nouns, which denote the real-world entities that the database will be modeling, and verbs, which denote the relationships between those entities. Figure 3-7 lists the nouns and verbs from Figure 3-6.

Nouns	Verbs		
university	university	*composed of*	departments
academic department	academic dept	*offers*	course
non-academic department	course	*has*	sections
course	student	*graduates in*	year
student	student	*majors in*	department
year	student	*enrolls in*	section
grade	student	*receives*	grades
section	professor	*assigns*	grades
professor	professor	*teaches*	section
car	section	*offered in*	year
campus	student	*requests*	permit
permit	student	*is issued*	permit
license plate	permit	*lists*	license plate
car model	permit	*lists*	car model

Figure 3-7: Extracting nouns and verbs from the requirements specification

The requirements specification defines the *scope* of the database. For example, it is clear from the requirements specification that the database is intended to hold data related only to student enrollments – there is no mention of employee issues (such as salaries, work schedules, or job descriptions), financial issues (such as departmental budgets, expenses, or student financial aid), or resource issues (such as classroom allocation, or the tracking of library loans).

Once we understand the scope of the database, we can cast a more critical eye on the nouns and verbs used in the requirements specification. Consider again Figure 3-7. The nouns *university* and *campus* are irrelevant to the database, because its scope includes only the one university and its campus. Similarly, the verb *composed of* is also irrelevant. The noun *non-academic department* is also irrelevant, because the database contains only academic information. The noun *car* is irrelevant, because the database holds information about the permit, not the car. And finally, the verb *requests* is irrelevant, because the database only needs to store the data for each issued permit, and not the data related to a requested permit.

Of course, a designer does not make decisions such as these by fiat. Typically, the designer takes the time to discuss and clarify the issues with the database users, so that everyone has the same understanding of the database's scope.

The designer uses the revised noun and verb list to construct a preliminary class diagram. Each noun maps to a class, and each verb maps to a relationship. There are no attributes yet. Figure 3-8 depicts the class diagram implied by the relevant nouns and verbs of Figure 3-7.

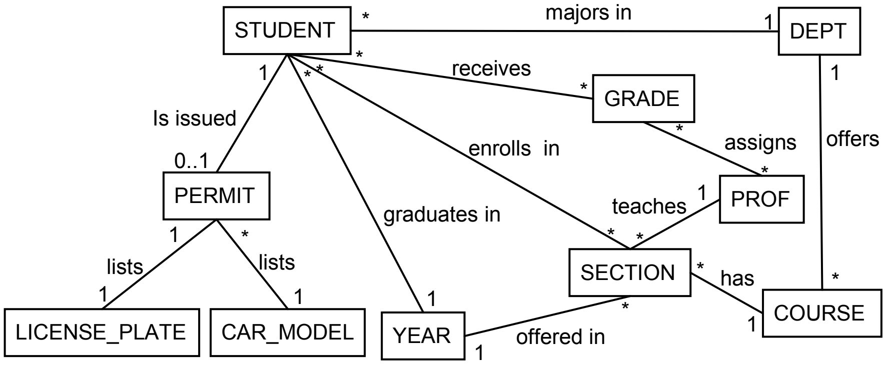


Figure 3-8: A preliminary class diagram based on the nouns and verbs of Figure 3-7

In order to make the class diagram more readable, we labeled each relationship with the verb that it denotes. You should convince yourself that the listed cardinalities correspond to your understanding of each relationship.

3.4.3 Inadequate relationships

The relationships in a class diagram depict how the entities in the database connect to each other. For example, suppose we are given a STUDENT entity. Then by following the *majors in* relationship, we can find the DEPT entity corresponding to that student's major. Or by following the *enrolls in* relationship, we can find all of the sections that the student has enrolled in. Relationships can also be composed. Starting with a STUDENT entity, we can find the courses that the student has enrolled in, by following the relationships *enrolls in* and *has*.

The designer must check relationships carefully to see if they convey their intended meaning. For example, suppose that we had mistakenly assumed that students enroll in courses instead of sections, and had therefore written the *enrolls in* relationship so that it went between STUDENT and COURSE (instead of between STUDENT and SECTION). See Figure 3-9.

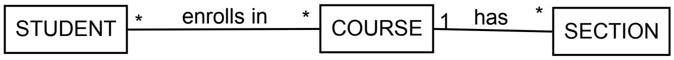


Figure 3-9: A class diagram containing an inadequate relationship

This class diagram has a problem. Given a student, we can determine the courses that the student enrolled in, and for each course, we can determine its sections. But we do not know which of those sections the student was in. We say that the *enrolls in* relationship in this diagram is *inadequate*.

Given a class diagram, there is no surefire way to tell if it contains an inadequate relationship. The only thing that the designer can do is to carefully examine each path through the class diagram to ensure that it expresses the desired information.

The diagram of Figure 3-8 has two inadequate relationships. Stop for a moment, look back at the figure, and see if you can tell what they are.

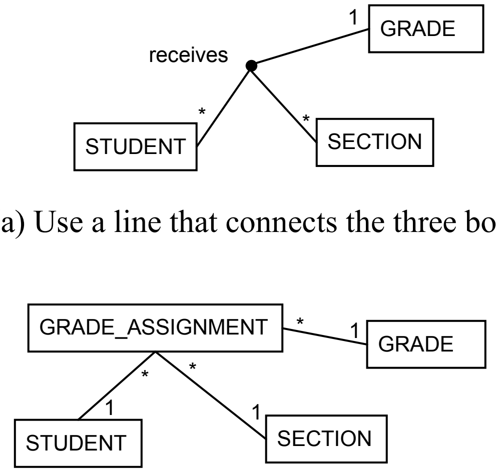
The two inadequate relationships are *receives* and *assigns*. Given a STUDENT entity, the *receives* relationship tells us what grades the student received, but there is no way to determine which section each grade was for. The *assigns* relationship is similarly problematic: it tells us which grades a professor gave out, but does not tell us to which student and in which section.

The cause of these two inadequate relationships was the simplistic way that we extracted them from the requirements specification. For example, if isn't enough to say "student receives grades". Instead, we need to say "student receives a grade for an enrolled section". Similarly, the *assigns* verb needs to be stated as "professor assigns grade to student in section".

These revised relationships are more complex than the other relationships we have seen, because they involve more than two nouns. They are often called *multi-way relationships*.

A multi-way relationship involves three or more classes.

There are two basic ways to represent a multi-way relationship in a class diagram. One way is to draw a line that connects more than two boxes. The other way is to turn the multi-way relationship into a class that has relationships to the participating classes. These two ways are depicted in Figure 3-10 for the *receives* relationship.



(a) Use a line that connects the three boxes

(b) Rewrite the relationship as a class

Figure 3-10: Two ways to handle a multi-way relationship

The annotations on the *receives* relationship in Figure 3-10(a) deserve explanation. The annotation next to GRADE is "1" because there will be exactly one grade for a student in a section. The annotation next to STUDENT is "*" because there can be many students in a section having the same grade. And the annotation next to SECTION is "*" because there can be many sections for which a student has the same grade.

In Figure 3-10(b), we replaced the relationship *receives* with the class GRADE_ASSIGNMENT. There will be a GRADE_ASSIGNMENT entity for each time a grade is assigned. (In terms of Figure 3-1, this class is the same as ENROLL, but we don't know that yet.)

These two approaches are notationally similar. For example, you can imagine the GRADE_ASSIGNMENT class "growing" out of the dot in the *receives* relationship. The approaches are also conceptually similar. The ability to turn a relationship into a class corresponds to the linguistic ability to turn a verb into a noun. For example, the verb "receives" can become "reception" or the gerund "receiving". The technical term for this process is *reification*.

Reification is the process of turning a relationship into a class.

In this chapter we shall adopt the second approach of Figure 3-10; that is, we choose to reify each multi-way relationship. There are two reasons. The first reason is that reification makes the class diagram easier to read, because multi-way relationships tend to be more awkward to decipher than regular binary relationships. The second reason is that the reified class provides more flexibility in later stages of the design process; for example, it can participate in other relationships and have its own attributes.

Returning to our running example, we also use reification to rewrite *assigns*, which is the other inadequate relationship in Figure 3-8. Here we need to create a new class that is related to STUDENT, SECTION, GRADE, and PROF. This new class will have one record for each grade assignment. Interestingly enough, we already have such a class, namely GRADE_ASSIGNMENT. We certainly don't need both classes in our diagram, so we combine them. The resulting class diagram appears in Figure 3-11.

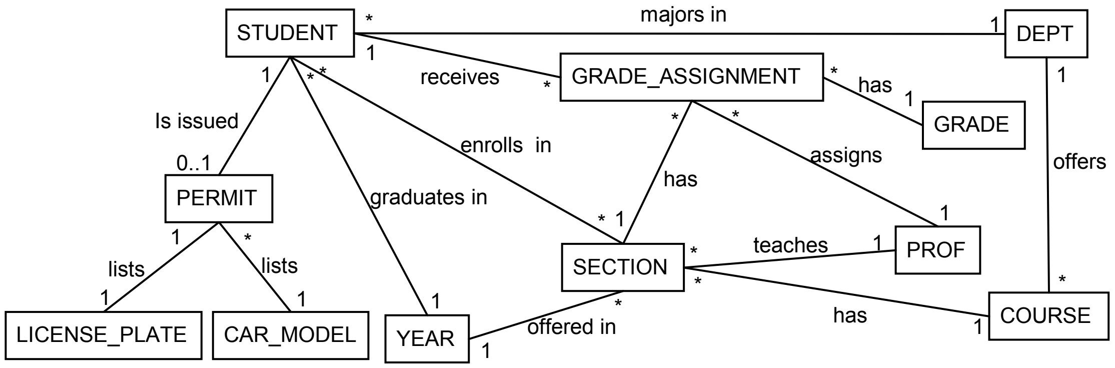


Figure 3-11: The class diagram that results from reifying the inadequate relationships

3.4.4 Redundant relationships

Consider Figure 3-11, and suppose I want to know the sections that a particular student has enrolled in. The most obvious tack is to follow the *enrolls in* relationship. However, with a bit of thought I can convince myself that the path from STUDENT to GRADE_ASSIGNMENT to SECTION has the same effect. In other words, the *enrolls in* relationship is *redundant*. The database doesn't need it. Removing the relationship from the class diagram will not change the content of the database.

> *A relationship is redundant if removing it does not change the information content of the class diagram.*

Redundant relationships are not desirable in a class diagram, for several reasons:

- *They are inelegant.* Ideally, each concept in a class diagram should have its own unique, proper spot.
- *They obfuscate the design.* A class diagram that is littered with redundant relationships is more difficult to understand.
- *They cause implementation problems.* If a user updates the database by modifying the contents of a redundant relationship without also modifying the non-redundant portion of the database (or vice versa), then there will be two different versions of the same relationship. In other words, the database will become inconsistent.

We express this concern as the following design rule:

> *Strive to eliminate redundant relationships from the class diagram.*

The relationship *assigns* (between PROF and GRADE_ASSIGNMENT) is another example of a redundant relationship. This relationship tells us, for each grade assignment, who assigned the grade. But if we look back at the requirements specification, we see that the grades for a section are always assigned by the professor of that section. Consequently, we can get the same information by following the path from GRADE_ASSIGNMENT to SECTION to PROF. Therefore, the *assigns* relationship is redundant.

The designer of a class diagram must check each relationship for redundancy. For example, consider the *majors in* relationship between STUDENT and DEPT. It will be redundant if it is equivalent to the path from STUDENT to GRADE_ASSIGNMENT to SECTION to COURSE to DEPT. That path, however, denotes the departments that offer the courses taken by a student, which is quite different from the department that the student is majoring in. Thus we can infer that *majors in* is not redundant.

Figure 3-12 depicts the class diagram that results from removing the two redundant relationships. Exercise 3.1 asks you to verify that none of the relationships in this figure are redundant.

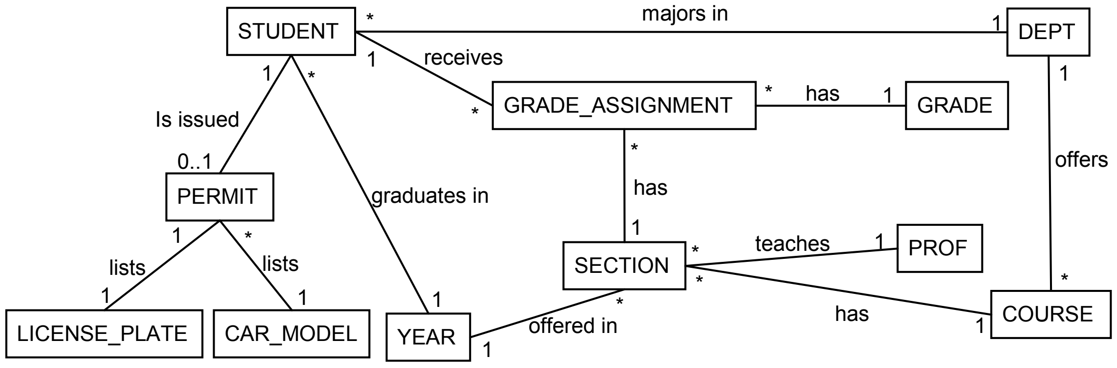


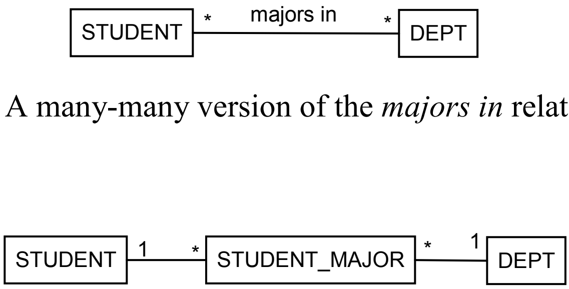
Figure 3-12: The class diagram that results from removing redundant relationships

3.4.5 Handling General Relationships

The transformation algorithm of Figure 3-4 only handles weak-strong relationships. A class diagram, however, may contain relationships with two strong sides or two weak sides. In this section we consider these possibilities, and show how they can be rewritten in a way that uses only weak-strong relationships.

Weak-Weak Relationships

Consider the *majors in* relationship between STUDENT and DEPT. This relationship is many-one in Figure 3-12, and asserts that each student must have exactly one major. Suppose instead that students at the university can have any number of majors; in this case, the annotation next to DEPT should be "*", resulting in the many-many relationship of Figure 3-13(a).



(a) A many-many version of the *majors in* relationship

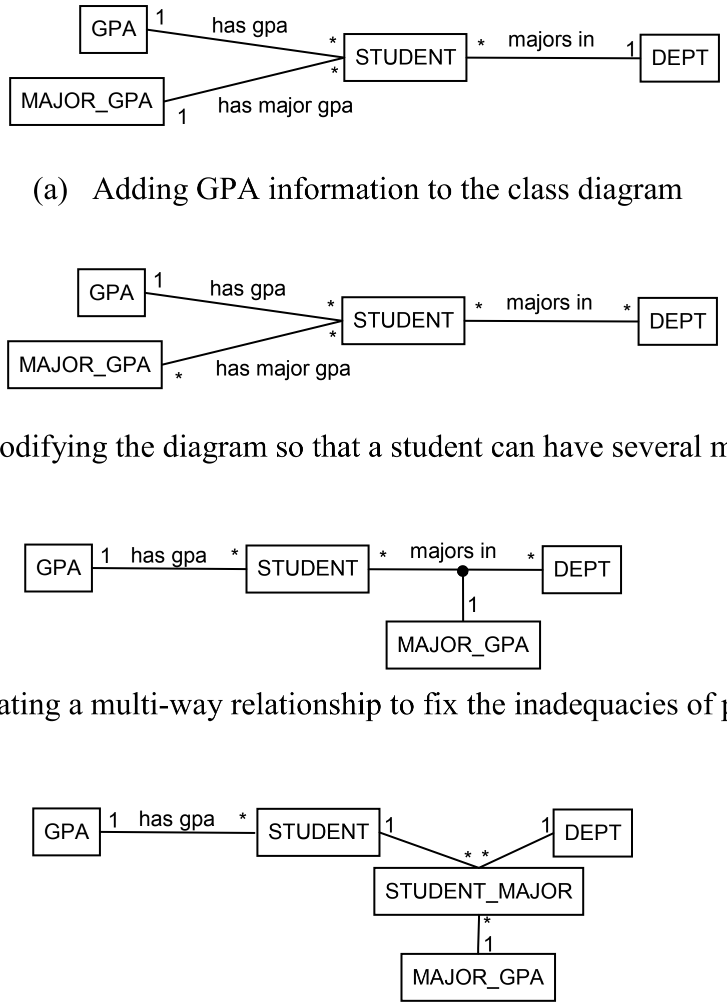
(b) Reifying the many-many relationship into two many-one relationships

Figure 3-13: Transforming a many-many relationship

Figure 3-13(b) shows what happens when we reify that many-many relationship. The new class STUDENT_MAJOR has a relationship with each of STUDENT and DEPT. There will be one STUDENT_MAJOR entity for each major declared by each student. That is, if Joe has 2 majors, and Amy and Max each have one major, then there will be 4 instances of STUDENT_MAJOR. Each student can thus have any number of related STUDENT_MAJOR entities, but each STUDENT_MAJOR entity corresponds to exactly one student. Thus the relationship between STUDENT and STUDENT_MAJOR is many-one. Similarly, the relationship between DEPT and STUDENT_MAJOR is also many-one.

In other words, reifying a many-many relationship creates an equivalent class diagram that no longer contains the many-many relationship. This useful trick allows us to get rid of unwanted many-many relationships. But it is more than just a trick. Most of the time, reifying a many-many relationship actually improves the quality of the class diagram.

For example, suppose that our class diagram needs to store both the GPA and the major GPA for each student (where the "major GPA" is the GPA of courses taken in the student's major). Figure 3-14(a) shows the class diagram in the case when a student has exactly one major. This diagram has no problems. But suppose now that a student can have several majors. Then the student will also have several major GPAs, which means that the *has major gpa* relationship must also be many-many. Figure 3-14(b) depicts this class diagram.



(a) Adding GPA information to the class diagram

(b) Modifying the diagram so that a student can have several majors

(c) Creating a multi-way relationship to fix the inadequacies of part (b)

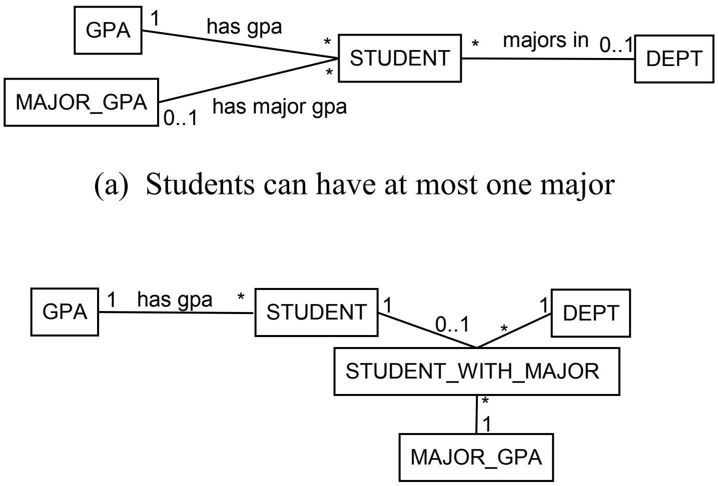
(d) The result of reifying the multi-way relationship

Figure 3-14: The need to reify a many-many relationship

The problem with Figure 3-14(b) is that the *has major gpa* relationship is now inadequate. Given a student, we can determine the various major GPAs, but we do not know which GPA corresponds to which major. The solution is to combine the *has major gpa* and *majors in* relationships into a single multi-way relationship, as shown in Figure 3-14(c). But now that we have created a multi-way relationship, we need to reify it anyway. The result is the new class STUDENT_MAJOR, as shown in Figure 3-14(d). Note how the MAJOR_GPA class winds up in a many-one relationship with STUDENT_MAJOR.

Reification can be used to remove all types of weak-weak relationships, not just many-many relationships.

For example, consider Figure 3-15(a), which shows the same diagram as before, except that we now suppose that students can have at most one major. The difference is that the annotation next to DEPT and MAJOR_GPA are now both "0..1".



(a) Students can have at most one major

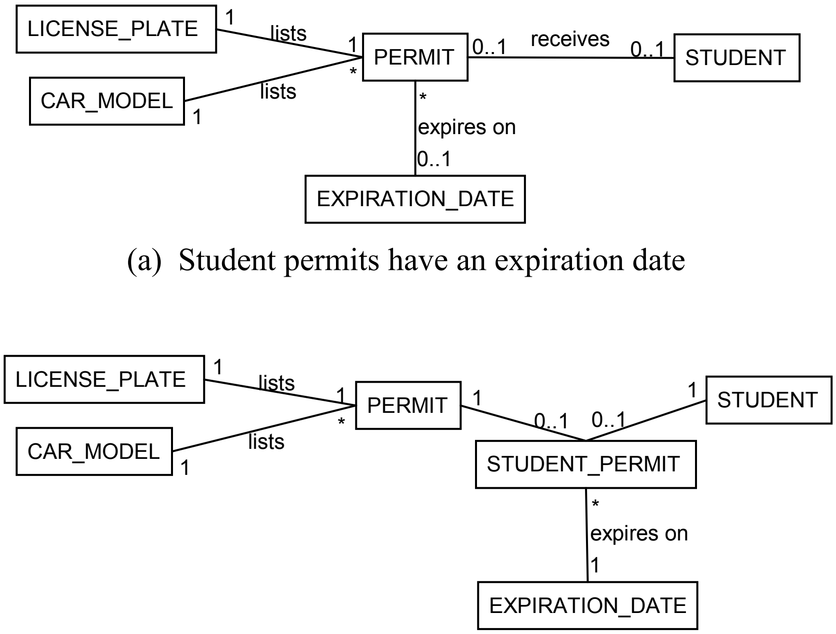
(b) The result of reifying the weak-weak relationship

Figure 3-15: Reifying a weak-weak many-one relationship

In this case, the relationship *has major gpa* is not inadequate (as it was in Figure 3-14), but it is problematic. The problem is that there is a dependency between *has major gpa* and *majors in* – namely, that a student should have a major GPA if and only if the student has a major. This dependency is inelegant, and indicates that they are really two aspects of the same multi-way relationship.

It thus makes sense to transform the class diagram in exactly the same way as before. We first merge the *majors in* and *has major gpa* relationships into a single multi-way relationship, and then reify that relationship. The result appears in Figure 3-15(b). Note the similarity of this class diagram with the analogous diagram of Figure 3-14(d); the difference is that the new class, which we call STUDENT_WITH_MAJOR, now has a "0..1" annotation in its relationship with STUDENT. This annotation reflects the fact that a student may have zero or one related STUDENT_WITH_MAJOR entity, but if there is one, then there is exactly one DEPT and MAJOR_GPA entity related to it.

The preceding two examples showed how to reify many-many and weak-weak many-one relationships. The final type of weak-weak relationship to consider is when both sides of the relationship have the "0..1" annotation. For an example of this type of relationship, consider the relationship between PERMIT and STUDENT in Figure 3-12, and suppose that the university gives out parking permits to staff as well as students. Moreover, suppose that student permits are "crippled" in the sense that they have an expiration date, whereas staff permits do not. In this relationship, each student can have at most one permit, but only some of the students will have a permit; in addition, each permit can be issued to at most one student, but only some of the permits will be issued to students. This relationship is symmetrical, in that both sides are annotated with "0..1". It is a weak-weak one-one relationship. The class diagram is shown in Figure 3-16(a).



(a) Student permits have an expiration date

(b) The result of reifying the weak-weak relationship

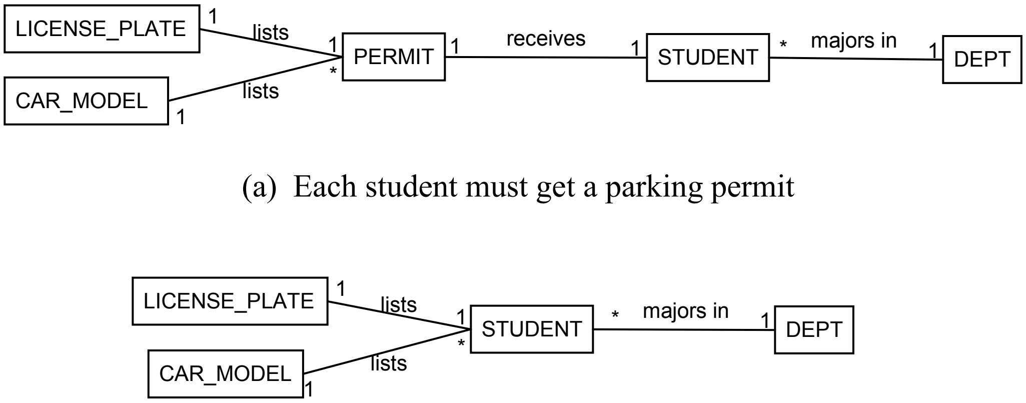
Figure 3-16: Reifying a weak one-one relationship

The problem with this diagram is the same as in the previous example. The relationship *expires on* is dependent on the relationship *receives*, and so they need to be combined into a single multi-way relationship. Figure 3-16(b) depicts the result of reifying that relationship.

The similarity of these three examples points out the essential nature of a weak-weak relationship, no matter what form it is in. A weak-weak relationship needs to be reified for two reasons. The first reason is a practical one: we do it because our transformation algorithm requires it. The second reason is conceptual: a weak-weak relationship often turns out to be a part of a multi-way relationship, and therefore will get reified anyway.

Strong-Strong Relationships

Recall that a strong-strong relationship has "1" annotations on both sides. In a strong-strong relationship, the entities in the two classes are in complete one-to-one correspondence. For example, the relationship between STUDENT and PERMIT would be strong-strong if the university required every student to get a parking permit. See Figure 3-17(a).



(a) Each student must get a parking permit

(b) Merging the PERMIT and STUDENT classes

Figure 3-17: Dealing with strong-strong relationships

One way to deal with a strong-strong relationship is to merge the two classes. Since the entities are in a one-one correspondence, they can be treated as different aspects of the same entity. Another way is to simply leave the relationship alone; when the time comes to transform the class diagram into a relational schema, we just treat one side of the relationship as if it were weak. This second way is important when one of the classes will be treated as an attribute of the other, as we shall see in the next section. Figure 3-17(b) illustrates both approaches. The classes PERMIT and STUDENT are merged, whereas the class LICENSE_PLATE is left alone.

3.4.6 Adding attributes to classes

Classes vs. Attributes

Up to this point, we have treated each noun as a class. It is now time to differentiate between two kinds of noun:

- nouns that correspond to an entity in the world, such as *student*;
- nouns that correspond to a value, such as *year*.

The first kind of noun corresponds to a class, and the second kind of noun corresponds to an attribute.

> *A class denotes an entity in the world.*
> *An attribute denotes a value that describes an entity.*

Actually, the distinction between *entity* and *value* is not as clear-cut as the above definition implies. For example, consider license plates. A license plate certainly is an entity in the real world. There are also many things that a database could store about a license plate: its state, its design, its physical condition, and so on. However, the requirements specification expresses a non-interest in these things – to it, a license plate is just a string of characters that is written on a permit. Consequently, it is reasonable to represent a license plate as an attribute in that database. A different database, such as a database maintained by the registry of motor vehicles, might choose to represent license plates as a class.

Now consider professors. The requirement specification doesn't say much about professors, so let's assume that we only need to store the name of each professor. But that does not necessarily mean that professors should be represented using an attribute. The question is whether the database should treat a professor's name as a random character string, or whether the database should maintain an explicit, "official" set of professor names. If the former case is true, then we want to represent professors using an attribute. If the latter case, then we need to keep PROF as a class.

Departments are in exactly the same situation as professors. We can choose to model departments as an attribute or as a class, depending on whether we want the database to explicitly keep track of the current departments.

The point is that the decision to model a noun as a class or as an attribute is not inherent in the real world. Instead, it is a decision about the relative importance of the noun in the database. A simple rule of thumb is this:

We should represent a noun as a class if we want the database to keep an explicit list of its entities.

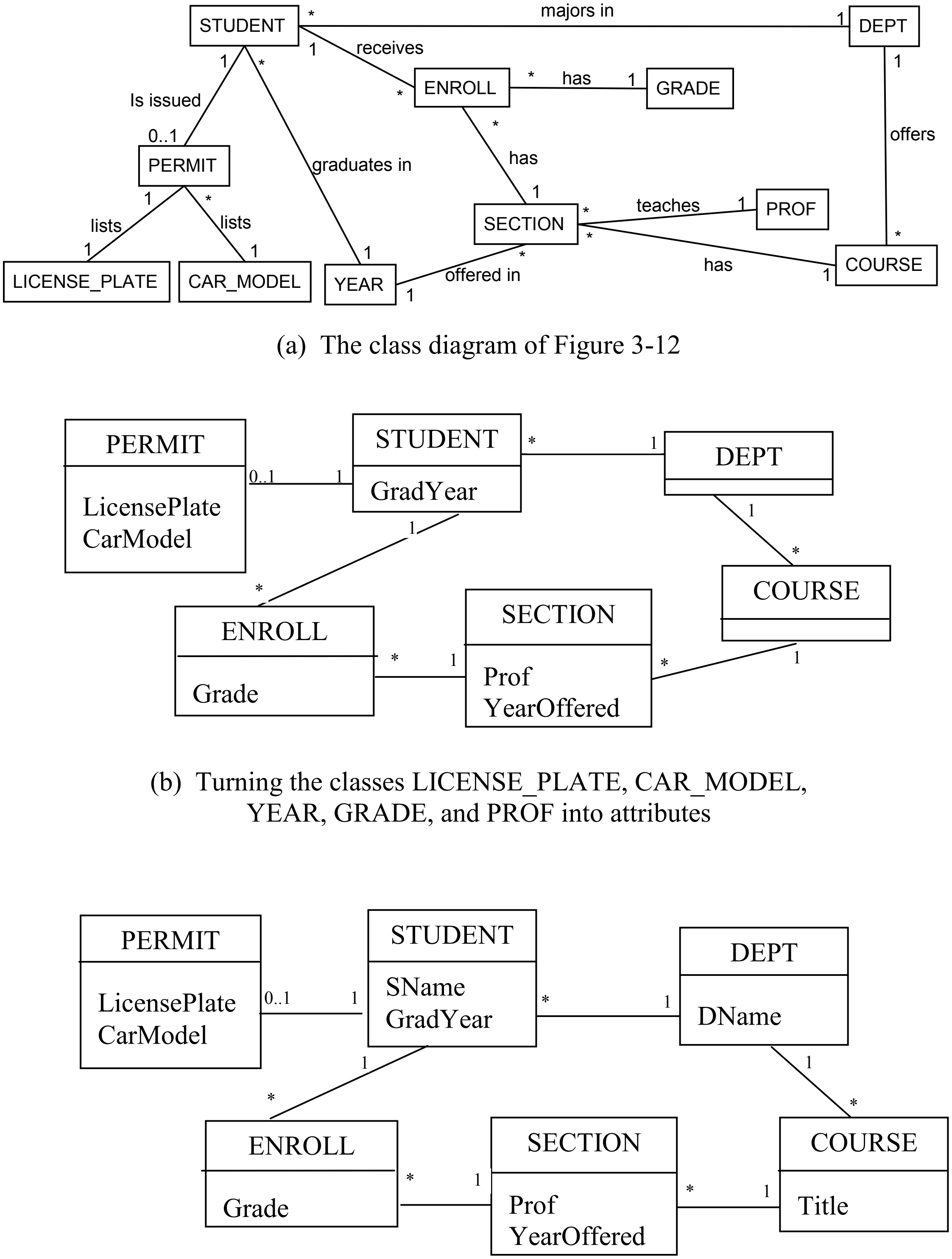
Transforming Classes into Attributes

Given a class diagram, the algorithm of Figure 3-18 shows how to transform one of its classes into an attribute.

1. Let C be the class that we want to turn into an attribute.
2. Add an attribute to each class that is directly related to C.
3. Remove C (and its relationships) from the class diagram.

Figure 3-18: An algorithm to turn a class into an attribute

For example, consider again the class diagram of Figure 3-12, which appears in Figure 3-19(a) (although for consistency with Figure 3-1, we have now renamed the class GRADE_ASSIGNMENT as ENROLL). We would like to turn GRADE, PROF, YEAR, LICENSE_PLATE, and CAR_MODEL into attributes.



(a) The class diagram of Figure 3-12

(b) Turning the classes LICENSE_PLATE, CAR_MODEL, YEAR, GRADE, and PROF into attributes

(c) Adding new attributes *SName*, *DName*, and *Title*

Figure 3-19: Adding attributes to a class diagram

The class GRADE is related only to ENROLL. Thus we simply need to add an attribute *Grade* to that class, and remove GRADE from the diagram. Similarly, an attribute *Prof* can be added to the class SECTION, and attributes *LicensePlate* and *CarModel* can be added to class PERMIT. The class YEAR is related to both STUDENT and SECTION. Thus we must add an attribute to each of these classes. Figure 3-19(b) depicts the resulting class diagram.

Adding Additional Attributes to the Diagram

The designer should also be on the lookout for useful attributes that are not mentioned in the requirements specification. Very often, the writer of a requirements specification assumes that the reader understands the basic concepts in the world to be modeled, such as the facts that students have names and courses have titles. Such concepts are typically not mentioned in the specification because they are too obvious to mention.

Once the designer has determined what classes will be in the class diagram, the designer can flesh out those classes with unstated, "obvious" attributes. In the example of Figure 3-19, we chose to add *SName* to STUDENT, *DName* to DEPT, and *Title* to COURSE. The resulting class diagram appears in Figure 3-19(c). You should note that this diagram is identical to that of Figure 3-1.

Attribute Cardinality

Attributes have cardinality, similar to relationships. An attribute can be *single-valued* or *multi-valued*. As the name implies, an entity will have one value for each single-valued attribute, and can have several values for each multi-valued attribute.

For example, consider again the class PROF in Figure 3-19(a). The *teaches* relationship is many-one, meaning that each section has a single professor. Consequently, the attribute *Prof* in Figure 3-19(b) will be single-valued. If *teaches* had been many-many, then *Prof* would be multi-valued.

Single-valued attributes are easy to implement in a relational database – each attribute corresponds to a field of a table. A multi-valued attribute can be implemented as a collection type (such as a list or array). Many relational systems support such collection types. If your target system does not, then it would be best to avoid multi-valued attributes – for example, by first reifying the many-many relationship.

Compound Attributes

In our discussion of the attribute-creation algorithm of Figure 3-18, we implicitly assumed that the transformed class had no attributes of its own. However, the algorithm actually works fine on classes that have attributes.

Consider for example Figure 3-19(c), and suppose that we decide that PERMIT should not be a class. The algorithm says that we should create an attribute *Permit* in STUDENT, but it is vague on what the value of that attribute should be. Clearly, the value should involve the values of *LicensePlate* and *CarModel* somehow. One possibility is to combine these two values into a single value; for example, the value of *Permit* could be a string of the form "corolla with plate #ABC321". Another possibility is to create a structured attribute value that contains the two underlying values. Such an attribute is called a *compound attribute*, because its value is a structure that consists of multiple sub-values.

For another example of a compound attribute, suppose that each student has an address, which consists of a street number, a city, and a postal code. We could give STUDENT three additional attributes, or we could create a compound attribute, *Address*, that is composed of these three sub-values. Compound attributes are often useful in queries. For example, a user can refer to an address either in its entirely, as a single unit; or the user can refer to the individual pieces separately.

Most commercial database systems support structured values, which means that a compound attribute can be implemented as a single field of a table. If your database system does not, then it is best to avoid compound attributes.

3.5 Relationships as Constraints

Each many-one relationship in a class diagram embeds a constraint. For example, the *major of* relationship implies that a student must have exactly one major department. When we transform a class diagram into a relational schema, we must be sure that the database system will be able to enforce the constraints implied by its many-one relationships.

As we have seen, a many-one relationship is implemented by a foreign key in the weak-side table, and referential integrity ensures that each weak-side record will have exactly one related strong-side record. So in general, there is no problem – referential integrity guarantees that the transformed relational database will satisfy each many-one constraint implied by the class diagram.

Actually, there is one area where a problem can arise. This problem area concerns the way that we reify multi-way relationships. Consider again Figure 3-10. The multi-way relationship *receives* is many-one, because there is exactly one grade for a given student and section. However, note that this constraint is missing from the reified class GRADE_ASSIGNMENT. Its relationships state that each grade assignment must have a student, section and grade, but there is nothing that prohibits a student from having two grade assignments for the same section. This constraint is also missing from the transformed relational schema of Figure 3-5(a), because several ENROLL records could have the same value for *StudentId* and *SectionId*.

The solution to this dilemma is to declare {*StudentId*, *SectionId*} to be the key of ENROLL, as in Figure 3-5(b). This key ensures that a student cannot enroll in a section more than once, and therefore cannot have more than one grade for a given section.

Generalizing this solution results in the following principle:

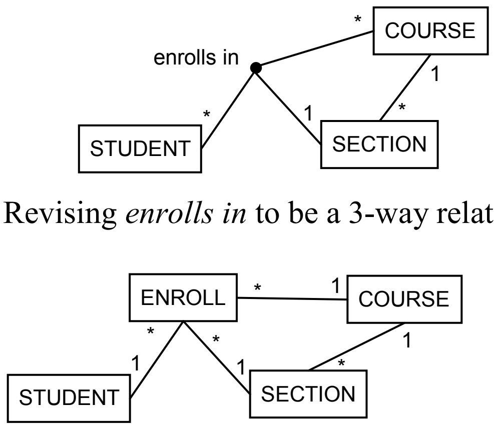
> *Let T be a table corresponding to a reified relationship.*
> *The key of T should consist of the foreign keys that correspond to*
> *the "many" sides of that relationship.*

This principle assumes, of course, that table T contains all of the necessary foreign keys, which implies that the class diagram must contain all of the relationships created by the reification. Unfortunately, this assumption will not be true if some of these relationships are redundant, and thus get eliminated from the class diagram. Consider the following example.

Suppose that the requirements specification for the university database contained the requirement shown in Figure 3-20(a). This requirement extends the relationship *enrolls in* of Figure 3-2. Whereas the relationship was originally just between STUDENT and SECTION, this new requirement forces the relationship to be multi-way, between STUDENT, SECTION, and COURSE. This relationship is depicted in Figure 3-20(b), along with the many-one relationship between SECTION and COURSE. Figure 3-20(c) shows what happens when we reify *enrolls in*, creating the new class ENROLL. And Figure 3-20(d) shows the tables that would be generated from these classes. We chose the key of ENROLL according to the above principle; that is, the fields{*StudentId*, *CourseId*} need to be the key of ENROLL in order to enforce the many-one constraint of the *enrolls in* relationship.

A student can enroll in a section of a course
only if the student has not taken that course before.

(a) The additional requirement



(b) Revising *enrolls in* to be a 3-way relationship

(c) The corresponding reified class ENROLL

```
ENROLL(StudentId, CourseId, SectionId)
SECTION(SectId, CourseId)
STUDENT(SId)
COURSE(CId)
```

(d) The relational schema generated from the class diagram

Figure 3-20: Adding a requirement that cannot be easily enforced

Note that the relationship between ENROLL and COURSE in Figure 3-20(c) is redundant; the same information can be obtained by going from ENROLL to SECTION to COURSE. However, removing that relationship from the class diagram causes the field *CourseId* to not be part of the ENROLL table, which means that we will not be able to enforce the many-one constraint.

This example illustrates a serious conundrum. How should we deal with the relationship between ENROLL and COURSE? On one hand, if we leave the relationship in the class diagram then the resulting ENROLL table will contain the redundant field *CourseId*, and the potential for inconsistent data. For example, consider Figure 3-20(d); there is nothing to stop a user from inserting a record into ENROLL that has the wrong course ID for its section. On the other hand, if we remove the relationship from the class diagram, then the resulting ENROLL table will not be able to enforce its key constraint. That is, there would be nothing to stop a user from inserting multiple records into ENROLL for the same student and course.

As far as anyone knows, this conundrum has no satisfactory resolution. The designer must choose whether to keep the redundant relationship and live with the potential for inconsistency, or to remove the relationship and deal with a possible constraint violation. For example, if we choose to keep the redundant relationship, then we can avoid inconsistency by forcing users to update ENROLL via a customized data-entry form; that form would not let the user specify a value for *CourseId*, but would automatically determine its the proper value from the SECTION table. Or if we choose to remove the redundant relationship, then we can enforce the key constraint via an explicit assertion (as we shall see in Figure 5-3).

3.6 Functional Dependencies and Normalization

3.6.1 Functional Dependencies

Creating a well-designed class diagram requires that we analyze each relationship carefully, in order to determine if any are redundant. As we have seen, the problem with redundant relationships is that they introduce redundant fields in the generated tables, and redundant fields can lead to inconsistent data.

However, it is not enough to focus on the class diagram. As we shall see, redundancy can creep into the generated tables in other ways too, such as via over-zealous attribute choices and by overlooked constraints. In this section we introduce the idea of *functional dependency* (or *FD*), and show how we can use FDs to check for redundancy in the generated tables.

> *An FD for table T is a constraint of the form* $A_1A_2...A_n \rightarrow B$
> *where B and each* A_i *are disjoint fields of T.*

The term “disjoint” means that all of the fields must be different. An FD cannot have its RHS field appear in the LHS, nor can the same field appear twice in the LHS.

An FD specifies the following constraint on its table: If two records have the same values for the LHS fields, then they must have the same values for the RHS field. If a table obeys this constraint, then we say that the FD is *valid* for the table.

> *An FD is valid for table T if all records that have the same values*
> *for the LHS fields also have the same value for the RHS field.*

This definition is very similar to the definition of superkey from Chapter 2. In fact, an FD is a generalization of a superkey. If K is a superkey of a table, then there cannot be two records having the same values for K, which means that the FD $K \rightarrow B$ will be valid for every other field B in the table.

As with any constraint, you cannot determine what FDs are valid by looking at the current contents of a table. Instead, you have to analyze the possible contents of the table. For example, here are some potential FDs for the SECTION table of Figure 1-1:

1. `Prof→CourseId` is valid if every record for a given professor has the same value for *CourseId*. In other words, it is valid if a professor always teaches the same course every year.

2. `Prof,YearOffered→CourseId` is valid if all of the sections that a professor teaches in a given year are for the same course.

3. `CourseId,YearOffered→Prof` is valid if all sections of a course during a year are taught by the same professor.

4. `CourseId,YearOffered→SectId` is valid if courses are taught at most once a year.

The set of FDs for a table can be used to infer its superkeys. For example, assume that *SectId* is a key of the SECTION table, and suppose that FD #4 above is valid. We can show that {*Course*, *YearOffered*} must be a superkey of SECTION. In proof, suppose not. Then there could be two records having the same values for {*CourseId*, *YearOffered*}. The FD then implies that those two records would have the same value for *SectId*, which is impossible because *SectId* is a key.

In general, suppose that $\{K_1,...,K_n\}$ is known to be a superkey and let *X* be some set of fields. Exercise 3.15 asks you to show that if the FD $X \rightarrow K_i$ is valid for each i, then *X* must also be a superkey.

Section 2.3 showed how a superkey can have superfluous fields. An FD can also contain superfluous fields, for similar reasons. If you add a field to the left side of an FD, then you get another FD whose constraint is less restrictive; so if the first FD is valid, then the second one is also. For example suppose that `Prof→Course` is valid, meaning that a professor always teaches the same course. Then the FD `Prof,YearOffered→Course` must also be valid, because it asserts that a professor teaches just one course in any given year. We say that an FD has a *superfluous field* in its LHS if removing that field results in a valid FD.

FDs having superfluous fields are not interesting. We therefore restrict our attention to *full FDs*:

A full FD is an FD that does not have superfluous fields.

3.6.2 Non-key-based FDs

Consider the LHS of a full FD. If these fields form a key, then we say that the FD is *key-based*.

A full FD is key-based if the fields in its LHS form a key.

Key-based FDs are essentially key specifications, and thus are a natural part of a relational schema. Non-key-based FDs, however, are quite different. In fact, they almost certainly result from a flaw in the database design.

A table that has a non-key-based FD contains redundancy.

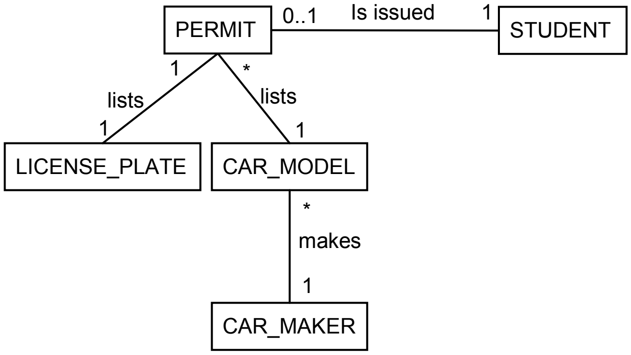
For example, suppose that the FD `Prof→Course` is valid for the table SECTION. Since the FD is not key-based, the table can contain several records having the same *Prof* value; these records will all have the same redundant value for *Course*.

There are several ways that non-key-based FDs can arise in a relational schema. We shall consider three common ones.

Choosing inappropriate attributes

A non-key-based FD can occur when a designer turns a class into an attribute. For an example, suppose that the requirements specification says that a parking permit should not only list the license plate and model or the car, but also its manufacturer. Figure 3-21(a) shows the revised class diagram. Note that the new class CAR_MAKER has a many-one relationship to CAR_MODEL, because every model has a specific manufacturer.

Now suppose that when the time comes to select attributes, we choose to create attributes for the license plate, car model, and car manufacturer, and add them to class PERMIT; see Figure 3-21(b). These three attributes seem perfectly reasonable; after all, the PERMIT table is almost the same as it was in Figure 3-5(a), and the new attribute *CarMaker* seems fairly innocuous.



(a) The revised class diagram

```
PERMIT(PermitId, LicensePlate, CarModel, CarMaker, StudentId)
```

(b) Choosing *LicensePlate*, *CarModel* and *CarMaker* to be attributes of PERMIT

```
PERMIT(PermitId, LicensePlate, CarModelId, StudentId)
CAR_MODEL(CarModelId, ModelName, Maker)
```

(c) Keeping CAR_MODEL as a class

Figure 3-21: Adding the car manufacturer to the database

However, the attribute *CarMaker* is actually problematic, because it causes the PERMIT table to have the non-key-based FD `CarModel→CarMaker`. And the presence of this FD means that the table will contain redundancy – Namely, any two permits for the same model will always have the same value for the field *CarMaker*. This redundancy, as always, can lead to an inconsistent database; for example, there is no way to stop Campus Security from issuing one permit for a "toyota corolla" and another permit for a "ford corolla".

A designer has two ways to deal with this redundancy. The first way is to ignore it. The rationale would be that although the FD is valid in the real world, it is totally uninteresting in the database. That is, the database really doesn't care about what models each car manufacturer makes. The purpose of the *CarMaker* field is simply to help identify a car, and an inconsistent value is not a big deal.

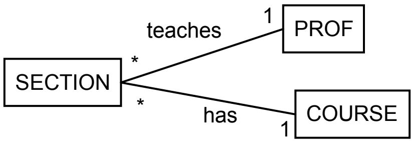
The second way is to enforce the FD via a foreign key. In order to do this, we need CAR_MODEL to be a class, not an attribute. The corresponding CAR_MODEL table will have a record for each car model, listing the model's name and manufacturer; see Figure 3-21(c). In this case, the CAR_MODEL table acts as the "official" list of all models; if a student has a car whose model is not on the list, then the list will need to be updated before the permit can be issued.

Omitted Relationships

A non-key-based FD can also arise when the designer omits a many-one relationship from the class diagram. Suppose for example that the university has the constraint that every professor teaches exactly one course. This constraint may not sound like a relationship, and so we may forget to include a relationship between PROF and COURSE in our class diagram. Figure 3-22(a) depicts our class diagram, and Figure 3-22(b) depicts its corresponding relational schema. It is not until we examine the schema for the table SECTION that we realize that the constraint applies and the non-key-based FD `Prof→CourseId` is valid.

The solution to this problem is to fix the class diagram and then regenerate the relational schema. In this case, we need to add a many-one relationship between PROF and COURSE. This new relationship causes the relationship between SECTION and COURSE to become redundant, and so we omit it. The resulting relational schema appears in Figure 3-22(c), and contains no non-key-based FDs.

Note how the relationship between PROF and COURSE means that we can no longer choose to turn PROF into an attribute; it must remain a class, in order to enforce the many-one constraint.



(a) The original class diagram

```
SECTION(SectId, Prof, CourseId)
COURSE(CourseId, Title)
```

(b) The generated relational schema

```
SECTION(SectId, ProfId)
PROF(ProfId, PName, CourseId)
COURSE(CourseId, Title)
```

(c) The generated schema after adding the omitted relationship
and removing the redundant relationship

Figure 3-22: Dealing with an omitted relationship between PROF and COURSE

Undetected Redundant Relationships

It is not easy to detect redundant relationships in a class diagram, especially when it is large. Consider again the class diagram of Figure 3-20(c) and its corresponding relational schema of Figure 3-20(d). In retrospect, we know that the relationship between ENROLL and COURSE is redundant, and can be omitted. But even if we miss detecting this redundant relationship in the class diagram, we still have the chance to discover it in the relational schema. In particular, the ENROLL table of Figure 3-20(d) has the non-key-based FD `SectionId→CourseId`. This FD identifies the redundant relationship for us. The solution to the problem is therefore to go back to the class diagram and remove the redundant relationship, if we desire.

3.6.3 Normal forms

A well-designed relational schema is said to be *normalized*. There are two relevant normal forms.

> *A relational schema is in Boyce-Codd Normal Form (BCNF)*
> *if all valid full FDs are key-based.*

> *A relational schema is in Third Normal Form (3NF)*
> *if the RHS field of all valid non-key-based FDs belongs to some key.*

The definition of 3NF is less restrictive than BCNF, because it allows a table to contain certain non-key-based FDs. In particular, suppose that the right side of a non-key-based FD *f* contains a key field. Then that FD would be allowed under 3NF, but not under BCNF.

This non-key-based FD *f* causes redundancy. If we remove the redundancy (say, by moving the fields of *f* to another table), then we will not be able to enforce the key. We saw an example of this situation in Figure 3-20, where f is the non-key-based FD `SectionId→CourseId`. The generated table ENROLL is not in BCNF, because of the FD. However, the table is in 3NF because the RHS side of that FD, *CourseId*, is part of the key of ENROLL.

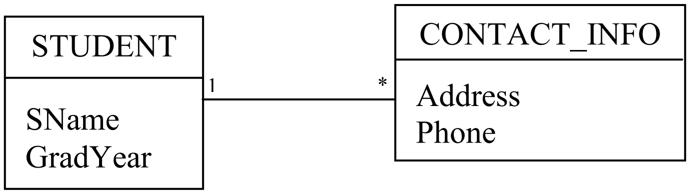
That example demonstrates that neither of the above definitions is perfect. The 3NF schema is "under-normalized", in the sense that the FD `SectionId→CourseId` is part of ENROLL and causes redundancy. On the other hand, the BCNF schema that results from removing the redundant relationship is "over-normalized", in the sense that the FD `StudentId,CourseId→SectionId` has been split between two tables, and must be enforced via a multi-table constraint.

The choice of which normal form to use basically comes down to a tradeoff between the conflicting FDs. How important is each FD? Will the potential inconsistencies of the 3NF schema be significant? How much do we care about enforcing the multi-table constraint in the BCNF schema? Fortunately, experience has shown that this dilemma occurs infrequently, and so the difference between 3NF and BCNF is usually irrelevant.

3.6.4 Normalized does not mean well designed

Given a relational schema, we examine it for non-key-based FDs in order to discover and fix flaws in its class diagram. Each non-key-based FD denotes a source of undesirable redundancy. However, a schema in normal form is not guaranteed to be well-designed, because not all instances of redundancy can be represented by FDs.

For example, suppose that we want our database to hold student email addresses and phone numbers. We decide to create a class CONTACT_INFO that contains the fields *Address* and *Phone*; see Figure 3-23(a). If a student can have several email addresses and several phone numbers, then it is not immediately obvious how these values should be stored in CONTACT_INFO records. We decide to use the following strategy: Each student will have a record for every combination of *Address* and *Phone* values. For example if Joe has two email addresses and three phone numbers, then we will store 2x3=6 records for him in CONTACT_INFO. See Figure 3-23(b).



(a)

CONTACT_INFO	**InfoId**	**Address**	**Phone**	**StudentId**
	91	joe@yahoo.com	6-1234	1
	92	joe@gmail.com	6-1234	1
	93	joe@yahoo.com	3-5555	1
	94	joe@gmail.com	3-5555	1
	95	joe@yahoo.com	9-9998	1
	96	joe@gmail.com	9-9998	1
	97	amy@gmail.com	6-4622	2
	98	amy@gmail.com	8-2323	2

(b)

Figure 3-23: Redundancy that FDs cannot catch

This design is amazingly bad. There is a lot of redundancy: Each email address is repeated as many times as the student has phone numbers, and vice versa. But this redundancy cannot be identified using FDs. The CONTACT_INFO table is in BCNF.

It turns out that this kind of redundancy corresponds to another kind of constraint, called a *MultiValued Dependency* (or MVD). If we desired, we could revise our normalization criteria to deal with MVDs. The resulting schema would be in something called *Fourth Normal Form* (4NF).

Unfortunately, there are other examples of redundancy that are not caught by MVDs. In the early years of the relational model, researchers developed increasingly more complex dependencies and normal forms, with each new normal form handling a situation that the previous normal forms could not handle. The goal was to be able to have a definition of "normal form" that always produced well-designed schemas[†].

That goal has been largely abandoned. Database design is still more of an art than a science. In the end, it is up to the database designer to craft an appropriate class diagram. Constructs such as FDs can help, but they cannot substitute for intelligent design and careful analysis.

3.7 Chapter Summary

- The information in a database is often vital to the functioning of an organization. A poorly designed database can be difficult to use, run slowly, and contain the wrong information.

- A good way to design a database is to construct a *class diagram*. A class diagram is a high-level representation of the database that focuses on what information the tables should hold and how they should relate to each other.

- A class diagram is built out of *classes* and *relationships*. A class denotes a table, and is represented in the diagram by a box. A relationship denotes (to some extent) a foreign-key connection between two tables, and is represented by a line connecting two boxes.

- Each side of a relationship has a *cardinality* annotation that indicates, for a given record, how many other records can be related to it. An annotation may be a "1", which indicates exactly one related record; a "*", which indicates any number of related records; or a "0..1", which indicates at most one related record. A relationship is said to be *many-many*, *many-one*, or *one-one*, depending on the annotations on each side.

- The "1" annotation is called a *strong annotation*, because it asserts that an entity cannot exist without a related entity. The other annotations are *weak annotations*. A foreign key corresponds to a weak-strong relationship.

- The algorithm of Figure 3-4 generates tables from a class diagram. This algorithm generates one table per class. The table contains a field for each of the class's fields, and a foreign key field for each relationship where the class is on the weak side.

[†] For the record, the Ph.D. thesis [Sciore, 1980] was a well-meaning but largely ineffective attempt in this direction.

- A relationship with two weak sides is called a *weak-weak relationship*. One example of such a relationship is a many-many relationship. The transformation algorithm does not generate tables for weak-weak relationships. Instead, we must first *reify* the relationship. We replace the relationship by a new class, which has weak-strong relationships to the other classes in the relationship. This new class can be thought of as "splitting" the weak-weak relationship. It improves the class diagram by making explicit a concept that was previously implicit.

- A *strong-strong* relationship has "1" annotations on both sides. Such a relationship denotes a single conceptual record that has been split up between the two classes, and can be removed by combining the two classes into a single class.

- A well-designed class diagram should contain no inadequate relationships, and no redundant relationships. The presence of an inadequate relationship means that the database will not be able to hold some desired data. The presence of a redundant relationship means that the database will contain the same fact in multiple places. Such redundancy can allow the database to become inconsistent.

- A *functional dependency* (FD) is a constraint that generalizes a superkey. An FD is *valid* if two records having the same LHS values must also have the same RHS value. An FD is *key-based* if its LHS values form a key.

- A table that contains a non-key-based FD violates the properties of a well-designed database. If all FDs are key-based, then the table is in *Boyce-Codd Normal Form* (BCNF).

- Removing a redundant field can make it impossible for a table to enforce a many-one constraint. If we want the database system to be able to enforce all such constraints, we can use *Third Normal Form* (3NF) instead of BCNF. 3NF is less restrictive than BCNF: A table in 3NF can have a non-key-based FD, provided that its RHS field belongs to some key.

- Database design is more of an art than a science. Normalization helps to find and fix problems in the class diagram, but it cannot guarantee a good diagram. In the end, the database designer needs to apply intelligent design and careful analysis.

3.8 Suggested Reading

The task of designing a database has three stages: first we construct the requirements document, then we model it using diagrams, and finally we map the diagrams to tables. This chapter focuses on the latter two issues. The first issue is the purview of Systems Analysis, and involves topics such as developing use-cases, interviewing users, consolidating and standardizing terminology, etc. The book [Hernandez 2003] provides an excellent introduction to this aspect of design. In addition, the book [Hay 1996] develops example diagrams for many common business situations, so that a designer can see how others have designed similar databases.

The book [Kent 2000] is a thought-provoking philosophical examination of the various ways that people conceptualize data. It argues that any given database can have more than one "correct" design, and makes it clear that database design is indeed far more difficult than we think.

This chapter uses UML as its diagramming language. Historically, designers have used many different diagramming languages, most notably variations on what are called *Entity-Relationship Diagrams*. The book [Teorey 1999] gives a comprehensive introduction to database design using these diagrams. The UML language was developed as a reaction to the proliferations of these language variants. UML stands for "Universal Modeling Language", and the hope was that it would become the standard design language, both for databases and for software. Of course, such high aspirations lead to feature creep, and UML has become very large. The diagrams used in this chapter are known as *class diagrams*, and comprise just one out of thirteen possible UML diagrams. A short introduction to all of UML appears in [Fowler and Scott 2003], as well as online at *www.agilemodeling.com/essays/umlDiagrams.htm*.

Our treatment of class diagrams in this chapter considers only the most common kinds of relationship. Other kinds of relationship can occur, such as subtypes and weak entities. These issues are considered in [Teorey 1999] and [Fowler and Scott 2003].

This chapter takes a somewhat narrow approach to normalization, primarily because it is our belief that most of normalization theory is irrelevant to modern design practice. A good overview of all of the normal forms and their purpose appears in [Jannert 2003].

3.9 Exercises

CONCEPTUAL EXERCISES

3.1 Verify that there are no redundant relationships in Figure 3-12.

3.2 Consider the online bookseller tables in Exercise 2.7. Draw a class diagram that corresponds to these tables.

3.3 Consider a relational schema. Suppose you use the algorithms of Figure 3-3 and Figure 3-4 to translate it to a class diagram and then back to a relational schema. Do you get the schema that you started with? if not, what is different?

3.4 Consider a class diagram, all of whose relationships are weak-strong. Suppose you use the algorithms of Figure 3-4 and Figure 3-3 to translate it to a relational schema and then back to a class diagram. Do you get the diagram that you started with? if not, what is different?

3.5 Consider the class diagram of Figure 3-11. Show that the relationship between STUDENT and GRADE_ASSIGNMENT is not redundant.

3.6 Transform the class diagram of Figure 3-12 to a relational schema (that is, without having chosen attributes). What fields do the tables have? How does your schema compare to the schema of Figure 2-1?

3.7 Suppose that we want the university database to contain the following information: The room and time where each section meets, the semester of each section (not just its year), the faculty advisor of a student, and the department to which each professor is assigned, and whether or not the student graduated.
a) Modify the class diagram in Figure 3-12 appropriately. Reify any weak-weak or multi-way relationships.
b) Choose appropriate attributes for the classes, and transform the resulting diagram to a relational schema.

3.8 Revise the class diagram of Figure 3-12 to handle each of the following assumptions. Reify any multi-way or weak-weak relationships.
a) Sections can be team-taught (that is, several professors can teach a section).
b) A department can offer multiple majors. (For example, the French department may offer a literature and a language major.)
c) A major can be interdepartmental (that is, offered by several departments).
d) A student can have multiple majors, and has an advisor for each major.
e) Each section has zero or more homework assignments. Each assignment has a string containing the text of the assignment, and a grade for each student.

3.9 Consider the class diagram of Figure 3-12, and suppose that each professor is assigned to a department. We therefore add a relationship to the diagram between PROF and DEPT.
a) What is the cardinality of this relationship?
b) The class diagram now has a cycle among the classes SECTION, COURSE, PROF, and DEPT. Explain why this cycle does not contain any redundant relationships.
c) Suppose we add the constraint that “a professor can only teach courses offered by her department”. Which relationships are now redundant?
d) Is it a good idea to remove the redundancy by deleting any relationships?
e) Explain why we could remove the redundancy if we merged the COURSE and PROF classes. Is this a good idea?

3.10 It is possible to reify a weak-strong relationship the same way that we reified weak-weak relationships.
a) Starting with the class diagram of Figure 3-12, give the class diagram that results from reifying the *majors in* relationship.
b) Is there a use for the reified class? Explain.

3.11 Suppose that the *teaches* relationship in Figure 3-19(a) is many-many. In the text, we stated that we could leave the relationship as it is, and transform the PROF class into a multivalued attribute of SECTION.
a) Give the class diagram that results from reifying the relationship instead. What attributes should get chosen?

b) What is the problem with choosing the reified class to be an attribute of SECTION?

3.12 Consider the class diagram of Figure 3-12.
a) For each class individually, rewrite the class diagram by turning that class into an attribute.
b) Rewrite the class diagram so that as many classes as possible are turned into attributes, without creating a non-BCNF table.
c) Rewrite the class diagram so that all classes except ENROLL become attributes. What does an ENROLL record look like in this case? What redundancies exist?

3.13 Does it make sense to choose attributes on a relationship-by-relationship case? That is, consider the class YEAR in Figure 3-12. Is it reasonable to transform YEAR into an attribute of STUDENT, but keep the class YEAR with its relationship to SECTION?

3.14 Give a reason why we might want GRADE to be a class instead of an attribute.

3.15 Show that if $\{K_1,...,K_n\}$ is a superkey and the FD $X \rightarrow K_i$ is valid for each i, then X must also be a superkey.

3.16 Translate the class diagram of Figure 3-11 to a relational schema.
a) The tables will have non-key-based FDs; what are they?
b) What problems can occur in thie schema?

3.17 Consider the SECTION table of Figure 1-1, and assume that it has no non-key-based FDs (that is, assume that the table is in BCNF). This table seems to have redundancy, because the fact “Turing taught course 12” appears in multiple records.
a) Explain why this redundancy cannot cause any inconsistencies.
b) Suppose that you want to ensure that such facts appear exactly once. How would you “split up” the SECTION table?
c) Compare your design from (b) with just having a single SECTION table.

3.18 Consider the database of Figure 1-1, but without the ENROLL table. We’ve decided not to use class diagrams. Instead, we feel that it is easy enough to simply store enrollment information in the STUDENT table.
a) Our first attempt is to add the fields *SectionId* and *Grade* to STUDENT, giving the following schema:

```
STUDENT(SId, SName, GradYear, MajorId, SectionId, Grade)
```

Note that *SId* is still the key of the table. Explain the problem with this schema.
b) We can fix the problem from part (a) by revising the key of STUDENT. What would a reasonable key be?
c) However, specifying this new key causes a different problem. Explain.
d) Give the non-key-based FDs in this table.

3.19 Consider the class diagram of Figure 3-1, except that the ENROLL and SECTION classes have been combined into a single ENROLL_SECTION class.

a) Explain the redundancies that would exist.
b) Give the non-key-based FDs that would be created by this merging.
c) Express the constraint "a student cannot take a course twice" as an FD for the generated ENROLL_SECTION table.
d) Is ENROLL_SECTION now in 3NF? Explain.

3.20 Suppose we add the following requirements to the specification of Figure 3-6:

- Professors always teach the same course every year.
- All sections of a course during a year are taught by the same professor.

a) Revise the class diagram of Figure 3-12 to embody these requirements.
b) Your answer to part (a) should have a redundant relationship that is necessary for the enforcement of a many-one constraint. Which relationship is it?
c) Without removing that redundant relationship, choose attributes for your class diagram and transform it to a relational schema.
d) Explain why your relational schema is in 3NF but not in BCNF.
e) Do you think that the 3NF schema is a good design, or would it be better to remove the redundancy and get a BCNF schema? Explain.

3.21 The database design of Figure 3-22 is really bad. Improve it.

PROJECT-BASED EXERCISES

The following exercises ask you to design databases for different applications. For each exercise, you should follow the design methodology completely: Create a class diagram, generate a relational schema from it, determine applicable FDs, and verify that the schema is in normal form.

3.22 Design a database to hold the CD and song information described in Section 3.1.

3.23 An insurance company wants to maintain a database containing the following information relating to its automobile policies:

> Each policy applies to exactly one car, but several people may be listed as drivers under the policy. A person may be listed under several policies. People are identified by their social-security number, and have names and addresses; cars are identified by their registration number, and have model names and years. Each policy has an identifying number and a list of coverage types (such as collision, comprehensive, damage to others' property, etc.); for each coverage type the policy has a maximum amount of coverage and a deductible. The database also contains information about accidents covered by each policy. For each accident, the company needs to know the car and driver involved, as well as the date of the accident and the total amount of money paid out.

3.24 A video-rental store wants to maintain a database having two major components: The first component lists general information about movies; the second component lists information about which movie videos the store carries, the store's customers, and what movies have been rented by those customers. The specifics are as follows:

- Each movie has a title, year, rating, running time, and director name. There may be several movies in the same year having the same title. A movie can be assigned to 0 or more genres (such as “comedy”, “drama”, “family”, etc.). Each actor has a name, birthyear and sex. An actor may appear in several movies, but will play at most one role in any movie.
- The store carries videos for some of these movies. For each video it carries, the store keeps track of how many copies it owns. A customer cannot rent a video without being a member. The store keeps track of the name, address, and membership number of each customer. The store also keeps track of every movie ever rented by each customer. For each rental, the store records the rental date and the return date. (The return date is null if the video is still being rented.)
- Customers can rate movies on a scale from 1 to 10. The ratings for each movie should be available, identified by which customer gave what rating. Similarly, the history of ratings for each customer should be available. (Should customers be able to rate only movies that they have rented? You decide.)

3.25 A hospital keeps a database on patients and their stays in the hospital. The specifics are:

- Patients are assigned a specific doctor that handles all of their all of their hospital stays.
- Each doctor has a name and an office phone number.
- For each patient visit to the hospital, the database records the date the patient entered, the date the patient left, and the room that the patient is staying in. If the patient is still in the hospital, then the “date left” field is null.
- The patient is charged per day that the patient spends in the hospital. Each room has a price per night.

3.26 The commissioner of a professional sports league has asked you to design a database to hold information on the teams in the league, the players who have played for the teams, the position that they played, and what the result of each game was. This database will hold information over many years, possibly back to the beginning of the league. The following constraints hold:

- A player can play only one position, and plays that position throughout his/her career.
- A player may play for several teams over his/her career. A player may change uniform numbers when traded to a new team, but keeps that number while with that team.
- Each game has a home team and an away team. There are no ties.

4
DATA MANIPULATION

In this chapter we focus on *queries*, which are commands that extract information from a database. We introduce the two query languages *relational algebra* and *SQL*. Both languages have similar power, but very different concerns. A query in relational algebra is *task centered*: it is composed of several operators, each of which performs a small, focused task. On the other hand, a query in SQL is *result oriented*: it specifies the desired information, but is vague on how to obtain it. Both languages are important, because they have different uses in a database system. In particular, users write queries in SQL, but the database system needs to translate the SQL query into relational algebra in order to execute it.

We also examine some other data-manipulation issues. We shall see how to write *update statements* that modify the database. And we shall also study *views*, which are tables whose contents are defined in terms of queries.

4.1 Queries

A user obtains information from the database by issuing a *query*. Queries in a relational database system have the following property:

> *The output of a relational query is a table.*

This property has three important features.

- *simplicity*: A user always accesses data by examining tables. If the user is not interested in any of the stored tables, then she simply writes a query to produce a table that does have the desired information.
- *efficiency:* Since the output of a query is a table, it can be saved in the database. The database system may then be able to use the saved output table to answer other queries.
- *power:* The output table of a query can be used as the input to another query, which allows users to create complex queries by combining smaller ones. Not only is it often easier to write a large query as a series of smaller ones, but these smaller queries can often be reused, which again increases the efficiency of the system.

The order of columns in a table is irrelevant to a relational query, because a query can only mention columns by their field name, not their position. Similarly, the order of rows in a table is also irrelevant, because a query operates equally on all records of its input tables. A user query can specify that the rows of the output table be shown in a particular order, but this order is for display purposes only; the number of rows and their content are not affected by the ordering.

In this book we shall look at two relational query languages: relational algebra, and SQL. SQL is the industry standard query language, and is the language of choice for most commercial database systems. On the other hand, relational algebra has no formal syntax, and current database systems do not support relational algebra as a user language. Instead, relational algebra is the foundation upon which real query languages like SQL are based. Every SQL query has a corresponding relational algebra query. This correspondence has two benefits:

- *Relational algebra is easier to understand than SQL.* If you are comfortable writing queries in relational algebra, then you will find it much easier to write in SQL.
- *Relational algebra is easier to execute than SQL.* Most database systems execute a user's SQL query by first transforming it into its relational algebra equivalent.

Section 4.2 covers relational algebra, and Section 4.3 covers SQL queries.

4.2 Relational Algebra

Relational algebra consists of *operators*. Each operator performs one specialized task, taking one or more tables as input and producing one output table. Complex queries can be constructed by composing these operators in various ways.

We shall consider twelve operators that are useful for understanding and translating SQL. The first six take a single table as input, and the remaining six take two tables as input. The following subsections discuss each operator in detail. For reference, the operators are summarized below.

The single-table operators are:

- *select*, whose output table has the same columns as its input table, but with some rows removed.
- *project*, whose output table has the same rows as its input table, but with some columns removed.
- *sort*, whose output table is the same as the input table, except that the rows are in a different order.
- *rename,* whose output table is the same as the input table, except that one column has a different name.
- *extend,* whose output table is the same as the input table, except that there is an additional column containing a computed value.
- *groupby,* whose output table has one row for every group of input records.

The two-table operators are:

- *product*, whose output table consists of all possible combinations of records from its two input tables.

- *join,* which is used to connect two tables together meaningfully. It is equivalent to a selection of a product.
- *semijoin*, whose output table consists of the records from the first input table that match some record in the second input table.
- *antijoin*, whose output table consists of the records from the first input table that do not match records in the second input table.
- *union*, whose output table consists of the records from each of its two input tables.
- *outer join,* whose output table contains all the records of the join, together with the non-matching records padded with nulls.

4.2.1 Select

The *select* operator takes two arguments: an input table, and a predicate. The output table consists of those input records that satisfy the predicate. A select query always returns a table having the same schema as the input table, but with a subset of the records.

For example, query Q1 returns a table listing those students who graduated in 2004:

```
Q1 = select(STUDENT, GradYear=2004)
```

Students who graduated in 2004

A predicate can be any Boolean combination of terms. For example, query Q2 finds those students who graduated in 2004 and whose major was either in department 10 or 20:

```
Q2 = select(STUDENT, GradYear=2004 and
                     (MajorId=10 or MajorId=20))
```

Students who graduated in 2004 with a major of 10 or 20

Recall that queries can be composed, by having the output table of one query be the input to another query. Thus queries Q3 and Q4 are each equivalent to Q2:

```
Q3 = select(select(STUDENT, GradYear=2004),
            MajorId=10 or MajorId=20)

Q4 = select(Q1, MajorId=10 or MajorId=20)
```

Students who graduated in 2004 with a major of 10 or 20
(two alternative versions)

In Q3, the first argument of the outermost query is another query, identical to Q1, which finds the students who graduated in 2004. The outer query retrieves, from those records, the students in department 10 or 20. Query Q4 is similar, except that it uses the name of Q1 in place of its definition.

A relational algebra query can also be expressed pictorially, as a *query tree*. A query tree contains a node for the tables and operators mentioned in the query. The children of a node denote its input tables. For example, the query tree for Q3 appears in Figure 4-1.

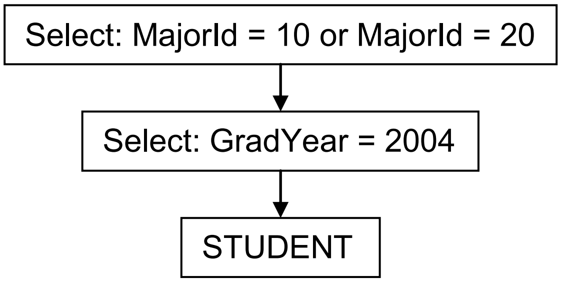


Figure 4-1: A query tree for Q3

4.2.2 Project

The *project* operator takes two arguments: an input table, and a set of field names. The output table has the same records as the input table, but its schema contains only those specified fields. For example, query Q5 returns the name and graduation year of all students:

```
Q5 = project(STUDENT, {SName, GradYear})
```

The name and graduation year of all students

Queries can be composed of both *project* and *select* operators. Query Q6 returns a table listing the name of all students majoring in department 10:

```
Q6 = project(select(STUDENT, MajorId=10), {SName})
```

The name of students having major 10

The query tree for Q6 appears in Figure 4-2.

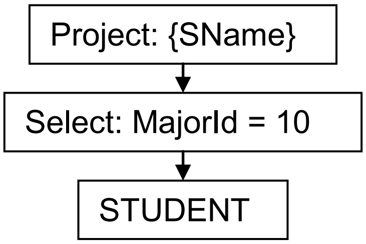


Figure 4-2: A query tree for Q6

Not all compositions of operators are meaningful. For example, consider the query you get by inverting Q6:

```
Q7 = select(project(STUDENT, {SName}), MajorId=10)
```

A meaningless query

This query does not make sense, because the output table of the inner query does not contain a *MajorId* field.

The output table of a project query may have duplicate records. For example, suppose that there are three students named "Pat" having major 10. Then the output of Q6 will contain *Pat* three times. If duplicate records are not desired, then they must be removed explicitly using the *groupby* operator, as we shall see in Section 4.2.6.

4.2.3 Sort

The *sort* operator takes two arguments: an input table, and a list of field names. The output table has the same records as the input table, but sorted according to the fields. For example, query Q8 sorts the STUDENT table by *GradYear*; students having the same graduation year will be sorted by name. If two students have the same name and graduation year, then their records may appear in any order.

```
Q8 = sort(STUDENT, [GradYear, SName])
```

Student records sorted by graduation year, then name

As mentioned in Section 4.1, the order of records in a table are for display purposes only, and do not affect the behavior of the other operators in the query. In fact, any operator following a *sort* operator is free to disturb the sort order of its output table. Thus the *sort* operator is typically the outermost operator in a relational algebra query.

4.2.4 Rename

The *rename* operator takes three arguments: an input table, the name of a field from the table, and a new field name. The output table is identical to the input table, except that the specified field has been renamed.

Renaming can be used to clarify the meaning of a field in the output table. For example, query Q6 returned the names of students majoring in department 10. Query Q9 renames the output field *SName* to be *CSMajors*, which is more informative:

```
Q9 = rename(Q6, SName, CSMajors)
```

Records from Q6, with field *SName* renamed to be *CSMajors*

This use of renaming is useful, but not particularly critical or compelling. However, when we examine multi-table operators we shall see that renaming is necessary, in order to alter the schema of one table to match (or not match) the schema of another. For example, the *product* and *join* operators require that the two input tables have the disjoint field names, and *rename* can be used to achieve this (as we shall see in query Q33).

4.2.5 Extend

The *extend* operator also takes three arguments: an input table, an expression, and a new field name. The output table is identical to the input table, except that it also contains a new field whose value is determined by the expression. For example, query Q10 extends STUDENT with a new field (called *GradClass*) that calculates the number of years since the first graduating class (which is, say, 1863):

```
Q10 = extend(STUDENT, GradYear-1863, GradClass)
```

Extending STUDENT to have the calculated field *GradClass*

Most database systems contain numerous built-in functions that can be used to perform interesting calculations. We shall discuss these functions in Section 4.3.2, in the context of SQL.

As an aside, it is worth remembering that the expression used to extend a table can be arbitrary, and need not mention fields of the table. For example, query Q11 extends STUDENT to have the new field *College*, whose value is always the string “BC”:

```
Q11 = extend(STUDENT, 'BC', College)
```

Extending STUDENT to have the calculated field *College*

4.2.6 GroupBy

The *extend* operator works horizontally, computing a new value from the field values of a single record. We now consider the *groupby* operator, which works vertically; it computes each of its values from a single field of a group of records. In order to understand how this operator works, we need to know two things:

- how records are placed into groups;
- how the computation is specified.

The groups are specified by a set of fields. Two records belong to the same group if they have the same values for each field of the set.

Computation is specified by calling an *aggregation function* on a field. For example, the expression *Min(GradYear)* returns the minimum value of *GradYear* for the records in the group. The available aggregation functions include *Min*, *Max*, *Count*, *Sum*, and *Avg*.

The *groupby* operator takes three arguments: the input table, a set of grouping fields, and a set of aggregation expressions. The output table consists of one row for each group; it will have a column for each grouping field and each aggregation expression.

For example, Query Q12 returns, for each student major, the minimum and maximum graduation year of students having that major.

```
Q12 = groupby(STUDENT, {MajorId},
                 {Min(GradYear), Max(GradYear)})
```

The maximum and minimum graduation year, per major

The table returned by Q12 appears in Figure 4-3, assuming the STUDENT table of Figure 1-1. Note that it has one row for each major ID, which means that *MajorId* will be a key of Q12. The table also has a field for each computed value, which are given default names. Each database system has its own way of choosing the field names for the aggregated values; typically, a user uses *rename* to rename those fields to something more appropriate.

Q12	MajorId	MinOfGradYear	MaxOfGradYear
	10	2004	2005
	20	2001	2005
	30	2003	2004

Figure 4-3: The output of query Q12

Query Q13 provides an example of a *groupby* query that contains two grouping fields:

```
Q13 = groupby(STUDENT, {MajorId,GradYear}, {Count(SId)})
```

The number of students in each major, per graduation year

The table returned by Q13 will have a row for each distinct (*MajorId*, *GradYear*) pair in the STUDENT table, and the computed value will indicate the number of students in each of these groups. Figure 4-4 depicts this output table, again assuming the STUDENT table of Figure 1-1.

Q13	MajorId	GradYear	CountOfSId
	10	2004	2
	10	2005	1
	20	2001	2
	20	2004	1
	20	2005	1
	30	2003	1
	30	2004	1

Figure 4-4: The output of query Q13

A *groupby* query may have an empty set of grouping fields. In this case, the entire input table is treated as a single group, and the output table will consequently have a single record. As an example, consider Query Q14:

```
Q14 = groupby(STUDENT, {}, {Min(GradYear)} )
```

The earliest graduation year of any student

This query returns a table containing one row and one column; its value is the earliest graduation year of any student.

A *groupby* query may have an empty set of aggregation functions. For example consider query Q15:

```
Q15 = groupby(STUDENT, {MajorId}, {} )
```

A list of the *MajorId* values of all students, with duplicates removed

This query returns a table containing one row for each value of *MajorId*, and only one column – namely, the column for that grouping field. In other words, the output table contains a row for each *MajorId* value in STUDENT, with duplicates removed. In general, a *groupby* query containing no aggregation is equivalent to a query that projects on the grouping fields and then removes duplicates.

Every aggregation function has two versions: a version that considers all values in the group, and a version that considers only the distinct values in the group. That is, the aggregation functions include *Count*, *CountDistinct*, *Sum*, *SumDistinct*, *Max*, *MaxDistinct*, etc. (Note that *MaxDistinct* is actually the same function as *Max*, but exists for the sake of completeness.) Consider the following two queries. Query Q16 returns the number of students having a major; query Q17 returns the number of different majors the students have.

```
Q16 = groupby(STUDENT, {}, {Count(MajorId)} )

Q17 = groupby(STUDENT, {}, {CountDistinct(MajorId)} )
```

The number of students having a major, and the number of different majors

Aggregation functions ignore null values. This fact is especially significant for the *Count* function, which counts the number of records having a non-null value for that field. Consequently, if the STUDENT table has no nulls (as in Figure 1-1), then it doesn't matter what field we count on. Consider Q16, and suppose we change the aggregation function to be either *Count(GradYear)* or *Count(SId)*; the output table would not change. In general, however, if a table has nulls then you need to be careful about which field you count. If you want to use the *Count* function to count the number of records in a group, then it makes most sense to always count the primary key field, because that field is guaranteed not to be null.

Groupby queries can be composed. Consider the problem of determining the most number of "A" grades that were given in any section. The information we need comes from the ENROLL table, and requires three processing steps:

- Select the records from ENROLL having a grade of 'A'.
- Count the records per section.
- Find the maximum count.

These steps comprise queries Q18-Q20. Figure 4-5 displays the output of these queries.

```
Q18 = select(ENROLL, Grade='A')
Q19 = groupby(Q18, {SectionId}, {Count(EId)} )
Q20 = groupby(Q19, {}, {Max(CountOfEId)} )
```

The most number of A's given in any section

Q18	EId	StudentId	SectionId	Grade
	14	1	13	A
	54	4	53	A
	64	6	53	A

Q19	SectionId	CountOfEId
	13	1
	53	2

Q20	MaxOfCountOfEId
	2

Figure 4-5: The output of queries Q18-Q20

Just for fun, Q21 does all three steps in a single query, and its query tree is depicted in Figure 4-6. Most people would agree that Q21 is unreadable (although its query tree isn't so bad). The moral is that complicated queries should always be broken up into smaller steps, if only for the sake of comprehension.

```
Q21 = groupby(groupby(select(ENROLL, Grade='A'),
                      {SectionId}, {Count(EId)} ),
              {}, {Max(CountOfEId)} )
```

The most number of A's given in any section (alternative version)

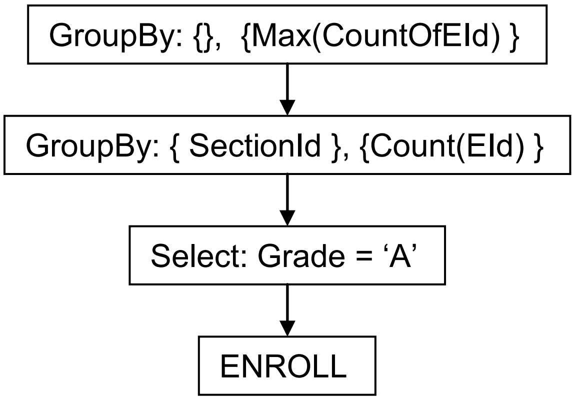

GroupBy: {}, {Max(CountOfEId) }
GroupBy: { SectionId }, {Count(EId) }
Select: Grade = 'A'
ENROLL

Figure 4-6: The query tree for Q21

You might have noticed that query Q21 (and Q20) determines the most number of "A" grades in a section, but does not tell us which section that is. The reason is that the outermost *groupby* operator must use the entire input table as its group; thus *SectionId* cannot be a grouping field, and therefore cannot be in the output table. In fact, it is impossible to write such a query using the six operators we have discussed so far (try it for yourself). The solution will be given in query Q31, after we introduce the *join* operator.

4.2.7 Product and Join

The operators we have discussed so far all act upon a single table. The *product* operator is the fundamental way to combine and compare information from multiple tables. This operator takes two input tables as arguments. Its output table consists of all combinations of records from each input table, and its schema consists of the union of the fields in the input schemas. The input tables must have disjoint field names, so that the output table will not have two fields with the same name.

Query Q22 returns the product of the STUDENT and DEPT tables:

```
Q22 = product(STUDENT, DEPT)
```

All combinations of records from STUDENT and DEPT

In Figure 1-1 there are 9 records in STUDENT and 3 records in DEPT. Figure 4-7 depicts the output table of Q22. This output table contains 27 records, one record for each pairing of a student record with a department record. In general, if there are N records in STUDENT and M records in DEPT, then the output table will contain N*M records (which, by the way, is the reason why the operator is called “product”).

Q22	SId	SName	MajorId	GradYear	DId	DName
	1	joe	10	2004	10	compsci
	2	amy	20	2004	10	compsci
	3	max	10	2005	10	compsci
	4	sue	20	2005	10	compsci
	5	bob	30	2003	10	compsci
	6	kim	20	2001	10	compsci
	7	art	30	2004	10	compsci
	8	pat	20	2001	10	compsci
	9	lee	10	2004	10	compsci
	1	joe	10	2004	20	math
	2	amy	20	2004	20	math
	3	max	10	2005	20	math
	4	sue	20	2005	20	math
	5	bob	30	2003	20	math
	6	kim	20	2001	20	math
	7	art	30	2004	20	math
	8	pat	20	2001	20	math
	9	lee	10	2004	20	math
	1	joe	10	2004	30	drama
	2	amy	20	2004	30	drama
	3	max	10	2005	30	drama
	4	sue	20	2005	30	drama
	5	bob	30	2003	30	drama
	6	kim	20	2001	30	drama
	7	art	30	2004	30	drama
	8	pat	20	2001	30	drama
	9	lee	10	2004	30	drama

Figure 4-7: The output of query Q22

Query Q22 is not especially meaningful, as it does not take into consideration the major of each student. This meaning can be expressed in a selection predicate, as shown in query Q23 and Figure 4-8:

```
Q23 = select(product(STUDENT, DEPT), MajorId=Did)
```

All students and their major departments

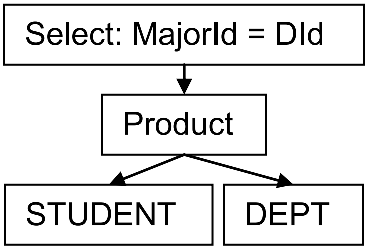


Figure 4-8: The query tree for Q23

The output table for this query contains only those combinations of records from the two tables that satisfy the predicate. Thus out of the 27 possible combinations, the only combinations that will remain are those for which the student's major ID is the same as the department's ID – in other words, the result table will consist of students and their major departments. Instead of 27 records, the output table now has only 9 records.

The combination of *select-product* operators in Q23 produces an output table in which related records are matched. This operation is called *joining* the two tables together, and the selection predicate is called the *join predicate*. Since joining is by far the most common way to connect tables together, we shall find it convenient to define the operator *join* as a shorthand for a selection of a product. Formally we have the following:

```
join(T1, T2, P) ≡ select(product(T1, T2), P)
```

Thus query Q24 is equivalent to Q23:

```
Q24 = join(STUDENT, DEPT, MajorId=DId)
```

All students and their major departments (alternative version)

In general, if you want to combine information from N tables, then you will need a query having N-1 join operators. For example, suppose we want to know the grades that Joe received in his courses during 2004. The query needs to connect tables STUDENT, ENROLL, and SECTION, and thus requires two join operators. We also need two selections, corresponding to the constants "joe" and "2004". Queries Q25-Q29 construct the query in small pieces; Figure 4-9 depicts the corresponding query tree.

```
Q25 = select(STUDENT, SName='joe')
Q26 = join(Q25, ENROLL, Sid=StudentId)
Q27 = select(SECTION, YearOffered=2004)
Q28 = join(Q26, Q27, SectionId=SectId)
Q29 = project(Q28, {Grade})
```

The grades Joe received during 2004

[redacted]

Figure 4-9: The query tree for Q25-Q29

Recall that the schema for a product (or join) query consists of the union of the fields of the input tables. Thus the output table for Q28 contains every field of the three joined tables. These joins were possible because all of these fields have different names; otherwise, we would have had to first use the *rename* operator.

The query tree of Figure 4-9 is not the only way to obtain the desired output table; there are many equivalent alternatives. The query tree will need to have two joins and two selects, but those operators can be arranged differently; for example, the selects can occur after both joins, and could be combined into a single selection predicate. Additional *project* operators could also be used to remove unneeded fields from intermediate queries. For example, query Q26 only needs the *SId* values of Q25; thus Q25 could be revised so that only Joe's ID is returned:

```
Q25a = project(select(STUDENT, SName='Joe'), {Sid})
```

An alternative version of Q25

Out of all these equivalent queries, which is best? At this point, we cannot say. In fact, determining the best query for a given problem turns out to be a difficult task, and requires knowledge of the size of the tables and the availability of indexes. Since users cannot be expected to perform the detailed computations, this responsibility is assumed by the database system. That is, users are encouraged to submit their queries without concern for efficiency; the database system then determines the most efficient version of the query. This process is called *query optimization*, and is the subject of Chapter 24.

Recall queries Q18-Q20, which computed the most number of “A” grades that were given in any section; their output tables appeared in Figure 4-5. In order to determine the section(s) having the most number of “A”s, we need to join the maximum value from Q20 to the section information of Q19, as follows:

```
Q30 = join(Q19, Q20, CountOfEId=MaxOfCountOfEId)
Q31 = join(SECTION, Q30, SectId=SectionId)
```

The sections that gave out the most A’s

Query Q30 joins Q19 with Q20; the output contains a record for each section whose count equals the maximum count. Query Q31 joins this output with the SECTION table, in order to return all information about those sections.

By far, the most common join predicate equates two fields; such a join is called an *equijoin*. Non-equijoins are somewhat obscure, and are expensive to compute. We shall ignore them in this book.

Moreover, the most common equijoin equates the key of one table with the foreign key of another. In terms of the class diagram, such a join corresponds to traversing a relationship between two classes. We call these joins *relationship joins*. All of the join examples we have seen so far are relationship joins. The remainder of this section considers two examples of more general joins.

Consider again the problem of retrieving the grades Joe received in 2004. Suppose that we change the problem so as to get the grades of the courses Joe took during his graduation year. The solution is essentially the same as in queries Q25-Q29; the difference is that instead of selecting on *YearOffered=2004*, we want to select on *YearOffered=GradYear*. And since the field *YearOffered* comes from SECTION and *GradYear* comes from STUDENT, this selection cannot occur until after both joins. The revised query tree appears in Figure 4-10.

[redacted]

Figure 4-10: The query tree to find the grades Joe received during his graduation year

The predicate *"YearOffered=GradYear"* in Figure 4-10 is a selection predicate that happens to equate two fields. The interesting thing about this predicate is that it could also be used as a join predicate, even though it doesn't correspond to a relationship. And if we did use it as a join predicate, then one of the other join predicates would be used as the selection predicate. The issue is that there are three predicates available to join three tables, but only two of the predicates can be used as join predicates. It doesn't matter which two we pick. This query is called a *circular join*, because the three join predicates link the three tables circularly. Exercise 4.6 asks you to rewrite this query using each of the possible combinations of two joins.

A non-relationship join also occurs when the query needs to compare records from the same table. Consider the problem of retrieving the students who have the same major as Joe. Queries Q32-Q34 give the solution:

```
Q32 = project(select(STUDENT, SName='joe'), {MajorId})
Q33 = rename(Q32, MajorId, JoesMajorId)
Q34 = join(Q33, STUDENT, JoesMajorId=MajorId)
```

The students that have the same major as Joe

Query Q32 returns a table containing Joe's major. The idea is to join this table with the STUDENT table, equating majors. Since Q32 contains only one record, the only STUDENT records matching it will be those students having the same major as Joe. A technical problem is that Q32 cannot be joined with STUDENT, because both tables have the field *MajorId*; the solution is to rename the field in one of the tables, as in Q33.

Query Q34 is called a *self join*, because it is joining the STUDENT table with (a portion of) itself.

4.2.8 Semijoin and Antijoin

Consider again query Q24, which joins tables STUDENT and DEPT. Its output table can be thought of as pairs of matching records – each STUDENT record is paired with its matching DEPT record, and each DEPT record is paired with its matching STUDENT records. Occasionally, a query doesn't care what the matching records are; it just wants to know if there is a match. The operator used for this purpose is called *semijoin*. For example, query Q35 lists those departments that have at least one student major:

```
Q35 = semijoin(DEPT, STUDENT, DId=MajorId)
```

The departments that have at least one student in their major

We read this query as "Return those records in DEPT for which there exist a matching STUDENT record."

Formally, the *semijoin* operator takes the same three arguments as a join: two input tables and a predicate. Its output table consists of those records from the first table that match some record in the second table. A semijoin is like a select, in that it returns a subset of the (first) input table. It differs from select in that its predicate depends on the contents of another table, instead of hardcoded constants.

A semijoin can always be implemented using a join. For example, queries Q36-Q37 are equivalent to Q35:

```
Q36 = join(DEPT, STUDENT, DId=MajorId)
Q37 = groupby(Q36, {DId, DName}, {})
```

The departments that have at least one student in their major
(alternative version)

The *groupby* operator in Q37 has a dual purpose. First, it causes the output table to contain only the fields of DEPT; and second, it removes the duplicates resulting from departments having more than one student major.

Because a semijoin can be implemented using a join, it is not a necessary operator. However, its simpler semantics often makes it preferable when it is applicable. For example, suppose we want to know which students have taken a course with professor Einstein. A solution is to first determine the sections Einstein taught, then determine the enrollment records corresponding to these sections, and then determine the students having those enrollments. Semijoins can be used effectively here, as shown in queries Q38-Q40 and the query tree in Figure 4-11:

```
Q38 = select(SECTION, Prof='einstein')
Q39 = semijoin(ENROLL, Q38, SectionId=SectId)
Q40 = semijoin(STUDENT, Q39, SId=StudentId)
```

The students who took a course with professor Einstein

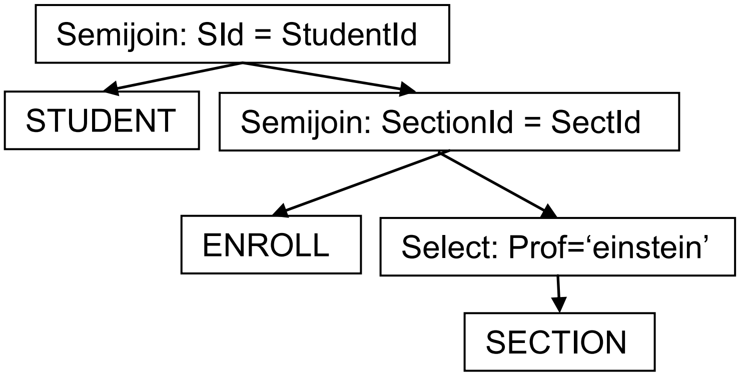


Figure 4-11: The query tree for Q38-Q40

The *antijoin* operator has the same arguments as semijoin. Its output table consists of those records from the first table that DO NOT match any record in the second table. For example, query Q41 returns those departments that have no student majors:

```
Q41 = antijoin(DEPT, STUDENT, DId=MajorId)
```

The departments that have no student majors

The antijoin can be thought of as the opposite of the semijoin, because it returns exactly those records that the semijoin omits. For example, consider queries Q35 and Q41. Note that their output tables have no records in common, and their union is exactly the DEPT table.

Although the antijoin and semijoin operators are clearly related, they have an important difference: *An antijoin cannot be simulated by a join.* In fact, the antijoin cannot be simulated by any combination of the operators we have seen so far. The issue is that a join (and a semijoin) finds matches; if there exists a record in the second table that matches a record in the first table, then the record will be in the output. But an antijoin is looking for the absence of a match, which is a very different situation.

For example, suppose we want to find the sections where nobody received a grade of “F”. We cannot simply select on the predicate “`Grade<>'F'`” in ENROLL, because a section might have failing as well as non-failing students. Instead, we need to make sure that *none* of the records for the section have a grade of “F”. The solution to this kind of problem is to specify the opposite of what we want, and then take the antijoin. In this particular query, we first identify those sections in which somebody received an “F”, and then antijoin those sections with the set of all sections. Queries Q42-Q43 perform this action:

```
Q42 = select(ENROLL, Grade='F')
Q43 = antijoin(SECTION, Q42, SectionId=SectId)
```

Sections where nobody received an F

The above query is an example of what is called a “not exists” query; it finds the sections where there does not exist a grade of “F”. The antijoin operator is the most natural way to write “not exists” queries.

Queries involving “for all” are also naturally written using antijoin, because “for all x” is the same as “not exists not x”. For example, to find those sections in which everyone got an “A”, we rephrase the problem as those sections in which there does not exist somebody who didn’t get an “A”. (Stop for a moment and slowly re-read the previous sentence. Keep re-reading it until it makes sense to you.) By rephrasing the problem this way, we see that the solution is the same as the above queries, except that we replace the selection condition in Q42 by “`Grade<>'A'`”.

As another example, queries Q44-Q48 find those professors who have never given an “F” in any section they have taught:

```
Q44 = select(ENROLL, Grade='F')
Q45 = semijoin(SECTION, Q44, SectionId=SectId)
Q46 = rename(Q45, Prof, BadProf)
Q47 = antijoin(SECTION, Q46, Prof=BadProf)
Q48 = groupby(Q47, {Prof}, {})
```

Professors who have never given a grade of F

This query is difficult to follow, so take it slowly. Queries Q44-Q45 find those sections where an “F” was given. Query Q46 renames the field *Prof* to *BadProf*, to indicate that the professors in this table are the ones we are not interested in. Query Q47 then returns those sections whose professor is not in the “bad list”. Finally, Q48 returns the names of the professors of those sections, with duplicates removed.

Although the concept of an antijoin is easy to define, it tends to be difficult to use. Many people find that the above query is very difficult to follow, and even harder to write. The issue seems to be that humans find it hard to keep track of the double negatives when turning “for all x” into “not exists not x”. Our only recourse is to not try to understand everything at once, but to take it step by step.

Queries involving “not exists” and “for all” may require several applications of the antijoin operator. For example, suppose we want to find those professors who gave at least one “F” in every section they taught. We can rephrase this as the professors who never taught a section where nobody got an “F”. Fortunately for us, query Q43 has already found the sections where nobody got an “F”, and so we can reuse it: First we label the professors who have taught one of those sections as “bad”; then we antijoin with the set of all professors.

```
Q49 = rename(Q43, Prof, BadProf)
Q50 = antijoin(SECTION, Q49, Prof = BadProf)
Q51 = groupby(Q50, {Prof}, {})
```

Professors who gave an F in every section they taught

Figure 4-12 gives the full query tree for this query, including the tree for Q43.

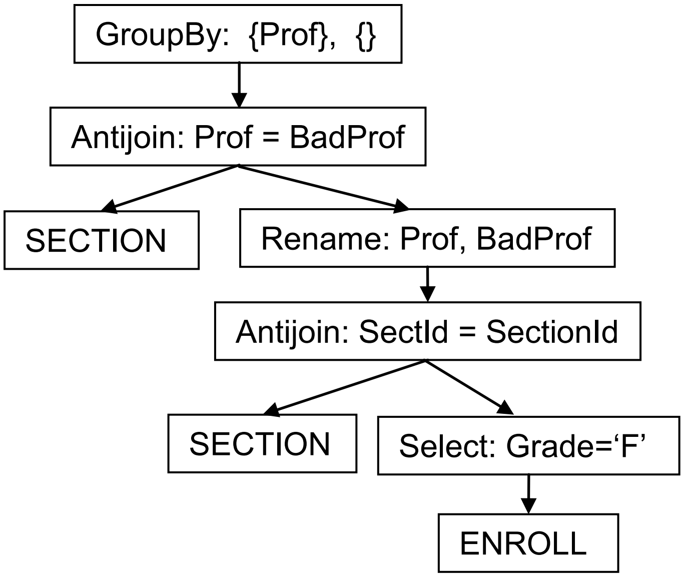


Figure 4-12: The query tree for Q49-Q51

4.2.9 Union and Outer Join

The *union* relational operator takes two arguments, both of which are tables; its output table contains those records that appear somewhere in the input tables. Because the *union* operator is taken from set theory, its output is a set – that is, a table without duplicate records. A union query requires that both underlying tables have the same schema; the output table will also have that schema.

The *union* operator is not used often in practice. One use is to coalesce similar values from different fields. For example, the output table of query Q54 contains the names of both students and professors, under the field name *Person*:

```
Q52 = rename(project(STUDENT, {SName}), SName, Person)
Q53 = rename(project(SECTION, {Prof}),  Prof,  Person)
Q54 = union(Q52, Q53)
```

The combined names of students and professors

The most common use of union is as part of the *outer join* operator. To understand what an outer join does, consider what happens when we join two tables, such as STUDENT and ENROLL. Those students who have not yet enrolled in any section will not appear in the join. The outer join operator adds these "missing" student records to the join, with null values for each field of ENROLL. Query Q55 expresses this outer join, and its output table appears in Figure 4-13.

```
Q55 = outerjoin(STUDENT, ENROLL, SId=StudentId)
```

The outer join of STUDENT and ENROLL

Q55	**SId**	**SName**	**MajorId**	**GradYear**	**EId**	**StudentId**	**SectionId**	**Grade**
	1	joe	10	2004	14	1	13	A
	1	joe	10	2004	24	1	43	C
	2	amy	20	2004	34	2	43	B+
	4	sue	20	2005	44	4	33	B
	4	sue	20	2005	54	4	53	A
	6	kim	20	2001	64	6	53	A
	3	max	10	2005	null	null	null	null
	5	bob	30	2003	null	null	null	null
	7	art	30	2004	null	null	null	null
	8	pat	20	2001	null	null	null	null
	9	lee	10	2004	null	null	null	null

Figure 4-13: The output of query Q55

Formally, the *outer join* operator takes the same arguments as the *join* operator. Its output table consists of the records of the join, unioned with the non-matching records from each of the tables. These non-matching records have null values for the fields of the other table.

For an example of an outer join query, suppose we want to know how many courses each student took. The information we need is in the ENROLL table; the simplest approach is to group by *StudentId* and count records, as in query Q56:

```
Q56 = groupby(ENROLL, {StudentId}, {count(EId)} )
```

Counting the number of enrollments for each student that has taken at least one course

The problem is that Q56 does not list students that took no courses. To get those students, we take the outer join with STUDENT before grouping and counting, as in queries Q57-Q58:

```
Q57 = outerjoin(STUDENT, ENROLL, Sid=StudentId)
Q58 = groupby(Q57, {SId}, {count(EId)} )
```

Counting the number of enrollments for all students

The non-matching student records in Q57 have null values for the fields of ENROLL. When grouped by *SId*, each of these records will form its own one-record group. The *count* operator will count a single null value for those groups (since *EId* is from ENROLL), and thus will return a zero for each of those students.

If the predicate of an outer join equates a foreign key with its primary key (and the foreign key does not contain nulls), then the non-matching records can come only from the primary-key table. Referential integrity ensures that every record from the foreign-key table has a matching record in the join. In terms of the above examples, we know that queries Q55 and Q57 can contain unmatched records from STUDENT, but they cannot possibly contain unmatched records from ENROLL.

4.3 SQL Queries

In relational algebra, a typical query is composed of several sub-queries, one for each of its operators. In contrast, an SQL query has multiple sections, called *clauses*, in which the various kinds of operations appear. As a consequence, SQL queries are compact – a single SQL query can specify many operations. In this section we shall cover the various clauses of a typical SQL query, and see how they encode the relational algebra operators.

This section describes some (but not all) features from the SQL standard. However, many commercial database systems do not support the entire SQL standard, and often include their own nonstandard extensions to SQL. If the database system you are using is behaving differently from the descriptions in this chapter, it might be necessary to check the user manual.

4.3.1 Basic SQL

Every SQL query must have at least a *select* clause and a *from* clause. The *from* clause specifies the input tables, and the *select* clause specifies the fields of the output table. For example, query Q59 returns the name and graduation year of all students:

```
Q59 = select STUDENT.SName, STUDENT.GradYear
      from STUDENT
```

The name and graduation year of all students

Note the prefixes in front of the field names. These prefixes serve to disambiguate fields having the same name in different tables. Within the query, fields are referenced by their prefixed form; but when the output table is constructed, the prefixes disappear. In other words, the input fields get renamed at the beginning of the query to have prefixes, and then get renamed at the end of the query to strip off the prefixes. Query Q59 is equivalent to the following series of relational algebra queries:

```
Q60 = rename(STUDENT, SName, STUDENT.SName)
Q61 = rename(Q60, GradYear, STUDENT.GradYear)
Q62 = project(Q61, {STUDENT.SName, STUDENT.GradYear})
Q63 = rename(Q62, STUDENT.SName, SName)
Q64 = rename(Q63, STUDENT.GradYear, GradYear)
```

The relational algebra equivalent to query Q59

This renaming is clearly pointless for Q59, but only because Q59 is such a simple query. This renaming can be quite useful (and even necessary) for more complex queries.

The prefix associated with a field name in an SQL query is called a *range variable*. By default, the name of the table is used as the range variable. A user can specify a different range variable by writing it in the *from* clause, after the table name. For example:

```
Q65 = select s.SName, s.GradYear
      from STUDENT s
```

The name and graduation year of all students (alternative version 1)

User-specified range variables have the advantage of being shorter and easier to write. As we shall see, they also allow an SQL query to use the same table twice, as in a self join.

SQL provides two shorthand notations that can make queries even more easy to write. The first shorthand is that "*" denotes all fields of a table; that is, "`t.*`" denotes all fields of the input table t. The second shorthand is that a field name can be written without a prefix if there is no ambiguity about what table a field belongs to. If an SQL query has only one input table, then there can be no possible ambiguity, and so prefixes need not be specified by the user (not even for "*"). For example, query Q65 is more commonly written as follows:

```
Q66 = select SName, GradYear
      from STUDENT
```

The name and graduation year of all students (alternative version 2)

It is important to realize that the prefixes still exist and the renaming still occurs in this query; it is just that the user doesn't have to specify it.

4.3.2 SQL types and their built-in functions

In Chapter 2 we introduced two SQL types: INT for integers, and VARCHAR for strings. We now investigate these (and some other) types in more detail, with an eye toward an understanding of how they can be used in a query.

The functions we shall discuss are part of the SQL standard. However, very few commercial database systems actually implement all of these functions; instead, they may implement similar functions having different syntax. If one of the functions mentioned here is not recognized by the system you are using, check its user manual.

Numeric Types

Numeric types in SQL come in two flavors: *exact* types and *approximate* types. An exact type supports exact arithmetic, whereas an approximate type allows for roundoff error. Numeric types have a *precision*, which is the number of significant digits it can represent. In exact types, the precision is expressed as decimal digits, whereas in approximate types, the precision is expressed as bits. Exact types also have a *scale*, which is the number of digits to the right of the decimal point. Figure 4-14 summarizes six common numeric types.

Type Name	Flavor	Precision	Scale	Sample Constant
NUMERIC(5,3)	exact	5 digits	3	31.416
INTEGER or INT	exact	9 digits	0	314159265
SMALLINT	exact	4 digits	0	3142
FLOAT(3)	approximate	3 bits	--	0.007
FLOAT	approximate	24 bits	--	3.1415926
DOUBLE PRECISION	approximate	53 bits	--	3.141592653589793

Figure 4-14: Numeric types in SQL

The fundamental exact type in SQL is *NUMERIC(precision,scale)*. For example, the type *NUMERIC(5,3)* has precision 5 and scale 3, and can therefore hold a value such as 31.416. The types INTEGER (or INT) and SMALLINT are special cases of NUMERIC that have a scale of 0. Their actual precisions are specified by each database system. Typically, SMALLINT values are represented using 2 bytes (resulting in a precision of 4 decimal digits) and INT values are represented using 4 bytes (for a precision of 9 decimal digits).

The fundamental approximate type in SQL is *FLOAT(precision)*. For example, FLOAT(3) can hold numbers that can be represented using 3 bits without considering scale, such as 0.007, 70, and 70,000. The types FLOAT and DOUBLE PRECISION have default precisions that are specified by each database system. The default precision for FLOAT is typically 24 bits, which is the maximum possible precision using a 4-byte representation. Similarly, the default precision for DOUBLE PRECISION is typically 53 bits, which is the maximum possible precision using an 8-byte representation.

The values in an arithmetic expression can be of any numeric type; the type of the result value is determined from the types of the inputs. In addition to the standard arithmetic operators, most database systems supply their own library of built-in functions that can be used to calculate numeric values. These functions are usually no different from those available in standard programming languages, and so we shall not cover them here.

One built-in function peculiar to SQL is *CAST*, which is used to transform a value from one type to any other type, if meaningful. When the destination type is numeric, this transformation will result in a truncated value if the scale of the destination type is too small, and an error if the precision is too small. The following three examples illustrate the possibilities:

```
CAST(3.14 AS INT)               returns 3
CAST(5.67 AS NUMERIC(4,1))      returns 5.6
CAST(12345.67 AS NUMERIC(4,1)) generates an error
```

String Types

In Chapter 2 we saw that the type *VARCHAR(n)* can hold strings of 0 to n characters. For example in the university database of Figure 1-1, student names are VARCHAR(10), department names are VARCHAR(8), and course titles are VARCHAR(30).

The type *CHAR(n)* specifies strings of exactly n characters – for example in the US, zip codes are CHAR(5), social-security numbers are CHAR(9) and phone numbers are CHAR(10).

Literal strings of either type are written by placing the characters within single quotes, and are case-sensitive. Standard SQL has many built-in string functions, the most common of which are described in Figure 4-15.

Function Name	**Meaning**	**Example Usage**	**Result**
lower (also, *upper*)	Turn the characters of the string into lower case (or upper case).	`lower('Einstein')`	`'einstein'`
trim	Remove leading and trailing spaces.	`trim('  Einstein   ')`	`'Einstein'`
char_length	Return the number of characters in the string.	`char_length('Einstein')`	`8`
substring	Extract a specified substring.	`substring('Einstein' from 2 for 3)`	`'ins'`
current_user	Return the name of the current user.	`current_user`	`'einstein'` (assuming that he is currently logged in)
||	Catenate two strings.	`'A. ' || 'Einstein'`	`'A. Einstein'`
like	Match a string against a pattern.	`'Einstein' like '_i%i_%'`	`true`

Figure 4-15: Some common SQL string functions

The *like* function warrants some discussion. The function is used as an operator; the left side of the operator is a string, and the right side is a pattern. A pattern can contain the special symbols '_' and '%'. The symbol '_' matches any single character. The symbol '%' matches any number of characters, including no characters. For example, consider the pattern `'_i%i_%'` from Figure 4-15. This pattern matches any string that contains a character, followed by 'i', followed by anything, followed by another 'i', followed by one or more characters; in other words, it matches a string whose second character is 'i' and contains another 'i' somewhere before the last character. In addition to 'Einstein', the following strings match this pattern:

```
'iiii'
'xiixxx'
'xixxixixxx'
```

A common use of patterns is to match all strings having a specified prefix. For example, the pattern ‘jo%’ matches all strings beginning with ‘jo’.

Date Types

The type *DATE* holds values that denote the year, month, and day of a date. A date literal is expressed as a string of the form ‘yyyy-mm-dd’, prepended by the keyword DATE. For example, `DATE '2008-08-04'` denotes July 4, 2008.

It does not make sense to add two dates, but it does make sense to subtract one date from another. The value you get by subtracting them is called an *interval*, and belongs to the type INTERVAL. The interval literal is expressed as follows: An interval of 5 days is written `INTERVAL '5' DAY`.

Figure 4-16 describes some common SQL functions that manipulate dates and intervals.

Function Name	Meaning	Example Usage	Result
current_date	Return the current date.	`current_date`	Today’s date
extract	Extract the year, month, or day from a date.	`extract(month, date '2008-07-04')`	`7`
+	Add an interval to a date.	`date '2008-07-04' + interval '7' month`	`date '2009-02-04'`
-	Subtract an interval from a date, or subtract two dates.	`date '2008-07-04' – date '2008-06-30'`	`interval '5' day`

Figure 4-16: Some common SQL date/interval functions

4.3.3 The *select* clause

In its most basic form, the *select* clause performs a *project* operation. It also can perform *extend* operations, via the notation “expression AS fieldname”. For example, the queries Q10-Q11 in Section 4.2.4 used the *extend* operator to calculate the graduating class and college of each student. Query Q67 combines them into a single SQL query:

```
Q67 = select s.*, s.GradYear–1863 AS GradClass, 'BC' AS College
      from STUDENT s
```

Calculating the graduating class and college of each student

The built-in functions of Section 4.3.2 can be used to calculate interesting output values. For example, query Q68 constructs a string-based identifier for each student and determines the decade in which they graduated:

```
Q68 = select 'Student #' || s.SId AS NewSId,
             s.SName,
             cast(s.GradYear/10 as integer) * 10 AS GradDecade
      from STUDENT s
```

Calculating a new ID and the graduation decade for each student

Figure 4-17 illustrates the result of this query.

Q68	NewSId	SName	GradDecade
	Student #1	joe	2000
	Student #2	amy	2000
	Student #3	max	2000
	Student #4	sue	2000
	Student #5	bob	2000
	Student #6	kim	2000
	Student #7	art	2000
	Student #8	pat	2000
	Student #9	lee	2000

Figure 4-17: The result of query Q68

The calculations in Q68 illustrate two points. First, note that a numeric ID value is used in the catenation operation, instead of the expected string value; SQL automatically performs the conversion from numeric to string. Second, note how the CAST function is used to truncate the result of the division, so that all graduation years in the same decade wind up having the same value.

Query Q69 uses the built-in date/interval functions to calculate the current number of years since a student graduated.

```
Q69 = select s.*,
          extract(YearOffered,Current_Date)-s.GradYear AS AlumYrs
      from STUDENT s
```

Calculating the number of years since graduation

The value of *AlumYrs* will be negative if the student has not yet graduated. Query Q70 uses this fact to determine whether a student has graduated.

```
Q70 = select q.*,
             case when q.AlumYrs>0 then 'alum'
                                   else 'in school'
                  end AS GradStatus
      from Q69 q
```

Determining whether a student has graduated

The value of the new field *GradStatus* is calculated using an SQL *case expression*. Case expressions tend to be awkward to use, which is why most vendors provide their own nonstandard version of an *if* function. For example, Q71 uses *if* to perform the same calculation as Q70:

```
Q71 = select q.*,
             if(q.AlumYrs>0, 'alum', 'in school') AS GradStatus
      from Q69 q
```

Determining whether a student has graduated (nonstandard alternative version)

A *select* clause can have arbitrarily many new fields. Since the clause is also used for projection, the new fields are also automatically placed into the output table, using the specified field name.

In terms of relational algebra, the *extend* operations specified by the AS keyword are preformed after the initial renaming but before the projection. For example, consider the contrived query Q72:

```
Q72 = select s.SName as X, s.SName, s.SName as Y
      from STUDENT s
```

A contrived but legal query

The output table of this query has three output fields, {*X, SName, Y*}, and the same number of rows as STUDENT. The value of *X, Y,* and *SName* will be the same in each row. It is equivalent to the query tree of Figure 4-18.

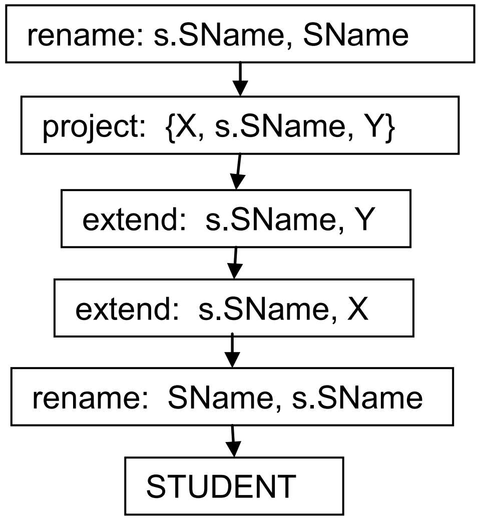


Figure 4-18: A query tree for Q72

4.3.4 The *from* clause

The *from* clause specifies the input tables of the query. Multiple tables correspond to the *product* operator. For example, query Q73 returns the 27 records in the product of STUDENT and DEPT:

```
Q73 = select s.*, d.*
      from STUDENT s, DEPT d
```

All 27 combinations of student and department records

Query Q73 contains six fields: four from STUDENT and two from DEPT. The specification of this *select* clause is legal because all of the field names are distinct. If they were not, then we would need to use the AS keyword to disambiguate. For example, query Q74 lists all 81 pairs of student names:

```
Q74 = select s1.SName as Name1, s2.SName as Name2
      from STUDENT s1, STUDENT s2
```

All 81 pairs of student names

Intuitively, we think of the AS keyword as performing a renaming here, although technically it is extending the table via a trivial computation and then projecting.

As we saw in Section 4.2.7, the most common way to combine tables is via a join (or outer join) operator. These specifications occur in the *from* clause. For example, query Q75 returns the names of students and their major departments:

```
Q75 = select s.SName, d.DName
      from STUDENT s join DEPT d on s.MajorId=d.DId
```

The join of STUDENT and DEPT

An outer join is specified by using "full join" instead of "join". As written, the output of Q75 omits two kinds of record: the DEPT records that have no student majors; and the STUDENT records having a null value for *MajorId*. If we revise the query to say "full join", then those records will be included.

To specify more than one join, use parentheses. Query Q76 returns the names of students and their professors:

```
Q76 = select s.SName, k.Prof
      from (STUDENT s join ENROLL e on s.SId=e.StudentId)
            join SECTION k on e.SectionId=k.SectId
```

The names of students and their professors (less readable version)

Clearly this syntax starts to get increasingly awkward as more tables are joined. As a result, people tend to specify joins using the *where* clause, as described next.

4.3.5 The *where* clause

An SQL query can optionally have a *where* clause, which holds the query's selection predicates. Query Q77 returns the name of all students in the class of 2005 or 2006:

```
Q77 = select s.SName
      from STUDENT s
      where (s.GradYear=2005) or (s.GradYear=2006)
```

The names of students graduating in 2005 or 2006

Recall that a join is equivalent to a selection of a product. Thus joins can be expressed by placing the input tables in the *from* clause (as the product), and placing the join predicates in the *where* clause (as the selection). That is, query Q78 is equivalent to Q76, and is perhaps more readable:

```
Q78 = select s.SName, k.Prof
      from STUDENT s, ENROLL e, SECTION k
      where s.SId=e.StudentId and e.SectionId=k.SectId
```

The names of students and their professors (more readable version)

The *where* clause will typically contain a mixture of selection predicates and join predicates. However, the user need not specify which predicates are used for joins and which for selections. Consider again the problem of retrieving the grades Joe received during his graduation year, whose solution appeared in Figure 4-10. The corresponding SQL query is given in Q79:

```
Q79 = select e.Grade
      from STUDENT s, ENROLL e, SECTION k
      where s.SId=e.StudentId and e.SectionId=k.SectId
      and s.SName='joe' and k.YearOffered=s.GradYear
```

The grades Joe received during his graduation year

Three of the terms in this *where* clause compare two fields for equality. As we saw earlier, only two of these terms can be join predicates, with the other one being a selection predicate. However, the query does not need to specify which ones are which. The choice of which predicates to use for joins is totally up to the system.

SQL built-in functions often show up in the *where* clause. Query Q80 returns the information about students whose names begin with ‘j’ and are graduating this year:

```
Q80 = select s.*
      from STUDENT s
      where s.SName like 'j%'
      and s.GradYear = extract(YEAR, current_date)
```

The students whose names begin with ‘j’ and graduate this year

If we assume that a professor’s login name is the same as the value in the *Prof* field, then query Q81 returns the students and grades for those sections taught by the currently logged-in professor:

```
Q81 = select s.SName, e.Grade
      from STUDENT s, ENROLL e, SECTION k
      where s.SId = e.StudentId and e.SectionId = k.SectId
      and  k.Prof = current_user
```

Grades given by the current professor-user

As a technical matter, the predicates in the *where* clause are evaluated after the *from* clause but before the *select* clause. Consequently, the *where* clause cannot reference any of the new fields created by the AS keyword.

4.3.6 The *group by* clause

Recall that the *groupby* operator has two components: the set of grouping fields, and the set of aggregation functions. In SQL, these two components are specified in different places: the grouping fields are in the optional *group by* clause; but the aggregation functions are in the *select* clause. For example, query Q82 finds the minimum and maximum graduation years per major, and is equivalent to Q12:

```
Q82 = select s.MajorId, min(s.GradYear), max(s.GradYear)
      from STUDENT s
      group by s.MajorId
```

The minimum and maximum graduation year, per major

The database system will assign a default field name to each of the two computed fields in this query. Default names are system-dependent and not always intuitive; thus you should always use the AS keyword to assign the output field name yourself.

A *groupby* query that has no grouping fields omits the *group by* clause. In this case, the entire input table will form a single group and the output table will consist of a single row.

The *group by* clause gets evaluated after the *where* clause but before the *select* clause. Consequently, it is not possible to have aggregation functions in the *where* clause, because the records have not yet been grouped. Similarly, the only non-computed fields allowed in the *select* clause are grouping fields, because the other input fields are removed during grouping.

To illustrate these limitations, consider the following example. Suppose we want to find the section that gave out the most “A” grades, as in Section 4.2.7. Queries Q83-Q84 compute the two aggregations, just like in queries Q19-Q20.

```
Q83 = select e.SectionId, count(e.EId) as NumAs
      from ENROLL e
      where e.Grade = 'A'
      group by e.SectionId

Q84 = select max(q.NumAs) as MaxAs
      from Q83 q
```

Counting the number of A’s given per section
and the maximum number of A’s given in any section

The issue at this point is how to associate the maximum number of A's of Q84 to its corresponding section ID. It is tempting to modify Q84 to add that ID value. There are several approaches that don't work; here are two of them:

```
Q84a = select q.SectionId, max(q.NumAs) as MaxAs
       from Q83 q

Q84b = select q.SectionId
       from Q83 q
       where q.NumAs = max(q.NumAs)
```

Two well-intentioned but syntactically illegal attempts
to find the section having the most A's

In Q84a, the hope is that the section ID having the maximum will somehow get associated with that maximum. But since grouping occurs before the *select* clause, the *SectionId* values will have been discarded by the time the *select* clause is reached; thus the query is illegal.

"OK", you say, "So why not modify Q84a to group by *SectionId*, thereby retaining the *SectionId* values?" Good idea – this modification, at least, makes the query legal. Unfortunately, its output isn't what we want. Since Q83 has already been grouped by *SectionId*, it has one record per section. Thus if we group again by that field, the groups won't change. Thus the query will calculate the maximum of each one-record group, instead of the maximum of all records.

Query Q84b seems intuitively reasonable. Unfortunately, it calls the aggregation function inside the *where* clause. SQL treats this as illegal because the aggregation function can be computed only when the records are grouped, and there is no way to understand Q84b in terms of grouping.

The solution, as in Section 4.2.7, is to first compute the maximum separately (as in Q84), and then determine the corresponding sections by doing a join. Query Q85 shows the correct approach.

```
Q85 = select Q83.SectionId
      from Q83, Q84
      where Q83.NumAs = Q84.MaxAs
```

The sections having the most A's

As a final example of the subtleties of the SQL *group by* clause, consider the problem of finding the name of each student and the number of courses they took. The most straightforward approach is to aggregate over the ENROLL table and then join with STUDENT, as in queries Q86-Q87:

```
Q86 = select e.StudentId, count(e.Eid) as HowMany
      from ENROLL e
      group by e.StudentId

Q87 = select s.SName, q.HowMany
      from STUDENT s, Q86 q
      where s.SId = q.StudentId
```

The name of each student and the number of courses taken
(incorrect version 1)

This query is not quite correct, because students who have taken no courses do not appear in the output. We need to do an outer join instead of a join. But if we revise Q87 so that it uses an outer join, then we will wind up with null values for *HowMany* instead of 0's (see Exercise 4.15).

A better approach is to do the outer join before counting, as in query Q88:

```
Q88 = select s.SName, count(e.Eid) as HowMany
      from STUDENT s full join ENROLL e
      on s.SId = e.StudentId
      group by s.SName
```

The name of each student and the number of courses taken
(incorrect version 2)

Query Q88 is almost correct. Its problem is that it groups records by student name instead of student ID, which means that any students having the same name will be grouped together and their count will be wrong. This query needs to group by *s.SId*. But if we group this way, we cannot put *s.SName* in the *select* clause. We seem to be stuck.

The solution to this conundrum is to group by both *SId* and *SName*, as in query Q89.

```
Q89 = select s.SName, count(e.Eid) as HowMany
      from STUDENT s full join ENROLL e
      on s.SId = e.StudentId
      group by s.SId, s.SName
```

The name of each student and the number of courses taken
(correct version)

Because *SId* is a key of STUDENT, the group of records for a particular *SId*-value will all have the same *SName*-value. Consequently, if we also group by *SName*, the groups do not change. In other words, it makes no logical sense to group by {*SId, SName*} instead of just {*SId*}; however, doing so allows SQL to retain *SName* values, which means they can be referenced in the *select* clause. This redundant grouping is perhaps a sneaky trick, but it is a very useful one.

Recall from Section 4.2.6 that you can remove duplicates from a query by creating a *groupby* query having no aggregation functions. The same idea holds in SQL. For example, query Q90 returns a list of all major departments of the class of 2004, with no duplicates:

```
Q90 = select d.DName
      from STUDENT s, DEPT d
      where s.MajorId=d.DId and s.GradYear=2004
      group by d.DName
```

The major departments of the class of 2004, with no duplicates

Looking at this query, it is difficult to realize that all we are doing is removing duplicates. Thus SQL allows the keyword DISTINCT to be used in the *select* clause instead. Query Q91 is equivalent to Q90:

```
Q91 = select distinct d.DName
      from STUDENT s, DEPT d
      where s.MajorId=d.DId and s.GradYear=2004
```

The majors of the class of 2004, with no duplicates (alternative version)

The keyword DISTINCT is also used with aggregation, to specify the functions *countdistinct*, *sumdistinct*, etc. For example, query Q17 returned the number of different majors that students have taken. Query Q92 shows how this query is expressed in SQL:

```
Q92 = select count(distinct s.Major)
      from STUDENT s
```

The number of different majors taken by students

4.3.7 The *having* clause

Suppose we want to find the professors who taught more than four sections in 2008. Query Q93 query is an incorrect (and in fact meaningless) attempt, because its *where* clause refers to the computed field *HowMany*. Since the *where* clause is always executed before the new fields are calculated, the field *HowMany* does not exist yet.

```
Q93 = select k.Prof, count(k.SectId) as HowMany
      from SECTION k
      where k.YearOffered=2008 and HowMany>4
      group by k.Prof
```

Professors who taught more than 4 sections in 2008 (incorrect version)

A solution to this problem is to use two queries, as in Q94-Q95. The first query performs the counting, and the second query tests for the value of *HowMany*.

```
Q94 = select k.Prof, count(k.SectId) as HowMany
      from SECTION k
      where k.YearOffered=2008
      group by k.Prof

Q95 = select q.Prof, q.HowMany
      from Q94 q
      where q.HowMany>4
```

Professors who taught more than 4 sections in 2008 (correct version 1)

Testing the value of an aggregation function is a fairly common task. The designers of SQL have therefore provided us with a clause, called the *having* clause, whose sole purpose is to let us combine the testing of Q95 with the grouping of Q94. The *having* clause contains a predicate. Unlike in the *where* clause, the predicate in the *having* clause is evaluated after the grouping has occurred, and therefore can call aggregation functions. Query Q96 is thus equivalent to Q94-Q95.

Note that although the predicate in the *having* clause can call aggregation functions, it cannot refer to the computed fields, because they have not yet been created.

```
Q96 = select k.Prof, count(k.SectId) as HowMany
      from SECTION k
      where k.YearOffered=2008
      group by k.Prof
      having count(k.SectId)>4
```

Professors who taught more than 4 sections in 2008 (correct version 2)

4.3.8 Nested queries

SQL uses *nested queries* to express semijoins and antijoins. For an example, consider Q35, which used a semijoin to determine the departments having at least one major. Query Q97 is equivalent:

```
Q97 = select d.*
      from DEPT d
      where d.DId in (select s.MajorId
                      from STUDENT s)
```

Departments that have at least one major

The keyword IN denotes set membership. The nested query on the right side of the expression is interpreted as returning a set of values; the expression returns *true* if its left-hand value is in that set. Note how the semijoin predicate "*DId = MajorId*" is encoded in the SQL query: The field from the outer table appears in its *where* clause, and the field from the inner table appears in its *select* clause.

In general, suppose that table T1 contains field A and table T2 contains field B. Then the correspondence between semijoins and nested queries is as follows:

```
   semijoin(T1, T2, A=B)  ≡   select * from T1
                              where A in (select B from T2)
```

A query tree having composed semijoins translates to an SQL query having multiply-nested queries. Figure 4-11 depicted a query tree that determined those students who took a course with professor Einstein. Query Q98 is equivalent:

```
Q98 = select s.*
      from STUDENT s
      where s.SId in (select e.StudentId
                      from ENROLL e
                      where e.SectionId in
                               (select k.SectId
                                from SECTION k
                                where k.Prof='einstein'))
```

Students who took a course with professor Einstein

This query can be expressed in SQL without nesting (see Exercise 4.16). However, many people prefer the nested version, because it more closely corresponds to how they think about the data. The limitation of nested queries is that they can only be used when the output table does not need any values from the nested table. That is, if we also want to know the grades of each student, then ENROLL cannot be nested.

Antijoin queries can be written using the NOT IN keyword. The correspondence is as follows:

```
 antijoin(T1, T2, A=B) ≡  select * from T1
                          where A not in (select B from T2)
```

For example, consider again queries Q42-Q43, which returned the sections where nobody received an “F”. Query Q99 is equivalent:

```
Q99 = select k.*
      from SECTION k
      where k.SectId not in (select e.SectionId
                             from ENROLL e
                             where e.Grade='F')
```

Sections where nobody received an F

4.3.9 The *union* keyword

The relational algebra operator *union* does not have its own SQL clause; instead, it appears as a keyword that combines other SQL queries, essentially the same as it does in relational algebra. For example, query Q54 returned a table that listed the union of student and professor names. The equivalent SQL query appears in query Q100:

```
Q100 = select s.SName as Person from STUDENT s
       union
       select k.Prof as Person from SECTION k
```

The combined names of students and professors

4.3.10 The *order by* clause

The *order by* clause contains the list of sort fields, separated by commas. By default, fields are sorted in ascending order; to sort a field in descending order, use the keyword DESC after the field.

The *order by* clause is evaluated after the *select* clause, which means that the sort fields are the fields of the output table, not the fields from the input table. Consequently, you are able to sort by computed fields, using the fieldname assigned to them via the AS keyword. And when you sort by non-computed fields, you omit the range variable prefixes (again, because we’re talking about fields from the output table).

For example, query Q101 returns the number of math majors in each graduation year, ordered by count descending; if multiple years have the same number of math majors, then those years are ordered by year ascending.

```
Q101 = select s.GradYear, count(S.SId) as HowMany
       from STUDENT s
       where s.MajorId = 20
       group by s.GradYear
       order by HowMany DESC, GradYear
```

The number of math majors per graduation year, sorted by count descending and then year

The *order by* clause is evaluated after all of the other clauses in an SQL query, even after unions. For example, query Q102 modifies Q100 to sort the output. The *order by* clause applies to the entire union query, not the second half of it.

```
Q102 = select s.SName as Person from STUDENT s
       union
       select k.Prof as Person from SECTION k
       order by Person
```

The combined names of students and professors, in sorted order

4.4 SQL updates

There are four basic ways to update a database in SQL:

- insert one new record by giving its values;
- insert multiple new records by specifying a query;
- delete records;
- modify the contents of records.

We shall discuss each in turn.

An *insert* statement is used to insert a single record into a table. The following statement inserts a new record into the STUDENT table:

```
insert into STUDENT (SId, SName, MajorId, GradYear)
values (10, 'ron', 30, 2009)
```

Inserting a record into STUDENT having explicit values

The list of field names and the list of values are parallel – each value is assigned to its corresponding field in the new record. An *insert* statement need not mention all fields in the table; the database system will assign a default value to any unmentioned field.

Instead of specifying an explicit set of values in an *insert* statement, you can specify a query. In this case, each record in the output of the query gets inserted into the table. For example, the following statements create a table ALUMNI and perform a bulk loading of records into it:

```
create table ALUMNI (Sid int, SName varchar(10),
                     Major varchar(8), GradYear int)

insert into ALUMNI (SId, SName, Major, GradYear)
   select s.SId, s.SName, d.DName, s.GradYear
   from STUDENT s, DEPT d
   where s.MajorId = d.DId
   and s.GradYear < extract(year, current_date)
```

Inserting records into the ALUMNI table

In this *insert* statement, a record gets inserted into the ALUMNI table for each output record of the query. Thus the table will contain a record for each student who has graduated prior to the current year.

We can also use this second form of *insert* to assign a mixture of values to a new record – values that are explicitly-given, as well as values from the database. For example, the following statement inserts a record into the STUDENT table for a new student Don, who has the same major and graduation year as student #1:

```
insert into STUDENT (SId, SName, MajorId, GradYear)
   select 22 as SId, 'don' as SName,
          s.MajorId, s.GradYear
   from STUDENT s
   where s.SId=1
```

Inserting a record for a new student
having the same major and graduation year as student #1

The query returns a single record. This record contains two constants (i.e. an *SId* of 22 and an *SName* of ‘don’), and two values from the query (i.e. the major ID and graduation year of student #1).

The *delete* statement deletes the records from a table that satisfy a predicate. For example, the following statement deletes all sections having no enrollments:

```
delete from SECTION
where SectId not in (select e.SectionId
                     from ENROLL e)
```

Deleting the sections having no enrollments

The *where* clause of a *delete* statement contains the deletion predicate. This predicate can be arbitrarily complex. In the above statement, the predicate uses a nested query to access information from another table. If the *where* clause of a delete statement is omitted, the statement deletes all records of the specified table.

The *update* statement modifies all records satisfying a specified predicate[†]. This statement is similar to *delete*, except that there is an additional clause that specifies what fields are to change, and how. For an example, the following statement changes the major and graduation year of all students in department 20:

```
update STUDENT
set   MajorId=10, GradYear=GradYear+1
where MajorId=20
```

Modifying the major and graduation year of students having major 20

The *set* clause contains a series of *assignments*, separated by commas. Although an assignment looks like a predicate, it is interpreted very differently. The left side of the assignment specifies the field that is to change, and the right side of the assignment is an expression that specifies the new value. In this example the first expression is the constant 10, and the second expression is the computation "*GradYear + 1*".

4.5 Views

4.5.1 Views and inline queries

So far we have been informal about how users name and execute queries. Formally, the naming of a query is always separate from its execution. Thus the statement that we saw earlier:

```
Q65 = select s.SName, s.GradYear
      from STUDENT s
```

defines the query named Q65, but does not execute it. Such a named query is called a *view*. Views are defined in SQL using the *create view* command. For example, view Q65 above would be defined in SQL as follows:

```
create view Q65 as
   select s.SName, s.GradYear
   from STUDENT s
```

Creating a view in SQL

[†] Unfortunately, the term "update" has two different meanings in the database community: In SQL, it denotes the specific command to modify a record; and in JDBC (as we shall see) it is a generic reference to any query that changes the database. In this book, I chose to side with JDBC, and therefore always refer to *update* as the SQL *modification statement*.

A user can refer to a view as if it were a table, either by requesting that its records be displayed, or by referencing it in another query or view definition. So to find the name of the students in the class of 2004, one could write the following query:

```
select q.SName
from Q65 q
where q.GradYear=2004
```

A query that uses the previously-defined view

This use of a view corresponds to the idea of composing queries in relational algebra – the output of Q65 is used as the input to the above query. It is also possible to express this composition directly, by using a query definition in the *from* clause:

```
select q.SName
from (select s.SName, s.GradYear from STUDENT s) q
where q.GradYear=2004
```

An equivalent query that defines a view in its *from* clause

The above two queries are completely equivalent. In fact, the inner query of the above query can be thought of as defining an implicit view, an unnamed version of Q65.

As we have seen, the best way to write a complex query is to split it into manageable subqueries, which are then composed. A user has the option of either creating views to hold the subqueries, or to write the subqueries explicitly in the *from* clause. Views have the advantage of being more readable, but require extra effort to create and destroy. Conversely, inner queries tend to be easier to write but become difficult to comprehend, especially as the number of inner queries grows.

4.5.2 Views and external schemas

Although views have value as a way to define subqueries, they also have a more important purpose. Recall the idea of logical data independence from Chapter 1, in which each user has a customized external schema and accesses the database exclusively through the tables in that external schema. The database system is responsible for mapping queries on the external schema to queries on the conceptual schema.

In a relational database system, the conceptual schema consists of the stored tables, and each external schema consists of views. For example, consider the external schema for the dean's office, given in Chapter 1:

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

Each of these tables can be defined as a view in SQL. The following view defines STUDENT_COURSES. (Exercise 4.17 asks you to provide the definition of STUDENT_INFO.)

```
define view STUDENT_COURSES as
   select e.StudentId as SId, k.YearOffered,
          c.Title, k.Prof, e.Grade
   from ENROLL e, SECTION k, COURSE c
   where e.SectionId = k.SectId and k.CourseId = c.CId
```

One of the tables in the external schema of the dean's office

This view table has the information on every course taken by each student, in a single, convenient place. By using this view, the dean's office can access the information it needs without having to figure out what query to write (and without actually having to write the query).

Views are ideal for implementing external schemas, because they are so simple to create and revise. If a user's requirements change, the DBA can easily rewrite an existing view or add a new view. If the stored tables need to be restructured, then the DBA can modify the view definition appropriately, without the user ever knowing that anything happened.

4.5.3 View Updates

When the database is being queried, views are indistinguishable from stored tables. When the database is being updated, however, views become troublesome. The problem, of course, is that the records in a view table don't really exist, because they are computed from other records in the database. Thus it makes no sense to manually insert, delete, or modify a view record. That being said, it sometimes is convenient to be able to change the database via a view.

We say that a view is *updatable* if every record r in the view has a unique corresponding record r' in some underlying database table. In this case, an update to virtual record r is defined as an update to stored record r'.

Single-table views are in general updatable, since each record of the view corresponds to a record from that table. A notable exception is when the view contains grouping, because an output record would then correspond to several underlying records.

Multi-table views are in general not updatable. The problem is that the virtual view record is composed from (at least) two stored records. It may not be possible to unambiguously translate an update operation on the virtual record into actual update operations on the underlying stored records. For example, suppose we define the following view, which lists the names of students and their majors:

```
create view StudentMajor as
   select s.SName, d.DName
   from STUDENT s, DEPT d
   where s.MajorId=d.DId
```

Is this view updatable?

Deleting the record [‘sue’, ‘math’] from this view means that Sue should no longer be a math major. There are at least three ways the database system could accomplish this deletion. For example: it could delete Sue’s record from STUDENT; it could delete the Math record from DEPT; or it could modify Sue’s *MajorId* value to give her a new major. Since the database system has no reason to prefer one way over the others, it must disallow the deletion.

Now suppose we want to insert a new record [‘joe’, ‘drama’] into the view. There are at least two ways to interpret this operation. Since Joe is currently in the computer science department, perhaps this insertion means that the *MajorId* value of his record should be modified. Or, since *SName* is not a key of STUDENT, perhaps we should insert a new STUDENT record having the name ‘joe’. There is no way for the database system to infer our intent.

Early versions of SQL did not permit updates to multi-table views. Over the years, people discovered that certain kinds of multi-table queries could be updatable. For example, consider again the deletion of [“sue”, “math”]. Most people’s intuition is that it makes most sense to delete the STUDENT record, because it is on the weak side of a many-one relationship. Standard SQL now specifies a set of conditions under which a multi-table view can be updated more or less intuitively.

The problem with allowing multi-table updates is that the database system will choose one action out of the many actions that are possible. If the user’s intuition does not correspond to that of the database system, then the user will be making unexpected changes to the database, which is almost certainly a recipe for disaster.

Another option that is becoming increasingly prevalent is to let users update the database via *stored procedures* instead of views. A stored procedure contains a series of database commands embedded in a conventional programming language. When a user executes a stored procedure, its commands are executed. For example, a stored procedure *DeleteStudentInfo* might perform the commands that the university thinks is appropriate for deleting a particular record from the STUDENT_INFO view, instead of what the database system might do by default.

4.6 Chapter Summary

- A user obtains information from the database by issuing a *query*.

- The output of a relational query is always a table. This property allows user to create large queries by composing smaller ones.

- A relational query operates on the contents of the database, not its physical implementation. In particular, the output of a query cannot depend on the column or record ordering of its input tables.

- Two important relational query languages are *relational algebra* and *SQL*.

- A relational algebra query is composed of *operators*. Each operator performs one specialized task. The composition of the operators in a query can be written as a *query tree*.

- The chapter describes twelve operators that are useful for understanding and translating SQL. The single-table operators are:
  - *select*, whose output table has the same columns as its input table, but with some rows removed.
  - *project*, whose output table has the same rows as its input table, but with some columns removed.
  - *sort*, whose output table is the same as the input table, except that the rows are in a different order.
  - *rename,* whose output table is the same as the input table, except that one column has a different name.
  - *extend,* whose output table is the same as the input table, except that there is an additional column containing a computed value.
  - *groupby,* whose output table has one row for every group of input records.

- The two-table operators are:
  - *product*, whose output table consists of all possible combinations of records from its two input tables.
  - *join,* which is used to connect two tables together meaningfully. It is equivalent to a selection of a product.
  - *semijoin*, whose output table consists of the records from the first input table that match some record in the second input table.
  - *antijoin*, whose output table consists of the records from the first input table that do not match records in the second input table.
  - *union*, whose output table consists of the records from each of its two input tables.
  - *outer join,* whose output table contains all the records of the join, together with the non-matching records padded with nulls.

- An SQL query has multiple sections, called *clauses*, in which the various kinds of operations appear. There are six common clauses: *select*, *from*, *where*, *group by*, *having*, and *order by*.

- The *select* clause lists the columns of the output table. An output column can be a column from an input table, or it can be the result of a calculation.

- The *from* clause lists the input tables. An input table can be the name of a stored table or view, or it can be an inline subquery that defines an anonymous view. The clause can also contain join specifications (both regular joins and outer joins).

- The *where* clause contains the selection predicate. The predicate is built out of expressions that can compare record values, and call built-in functions and operators. The operators *in* and *not in* test to see if a value is (or is not) a member of a set. This set can be defined by a *nested query*.

- The *group by* clause specifies the fields used for aggregation. The output table will consist of one row for every combination of field values that satisfies the selection predicate.

- The *having* clause works in conjunction with the *group by* clause. It contains a predicate that gets evaluated after the grouping occurs. Consequently, the predicate can call aggregation functions and test their values.

- The *order by* clause specifies the sort order for the output records. Sorting occurs after all other query processing is performed.

- SQL has four kinds of update statement: two forms of insert, a delete statement, and a statement to modify records.

- A *view* is a named query. Views are defined in SQL using the *create view* command.

- A user can refer to a view as if it were a table, either by requesting that its records be displayed, or by referencing it in another query or view definition.

- Views are ideal for implementing external schemas, because they can be easily created and revised. If user requirements change, the DBA can modify the view definitions, without impacting either the conceptual schema or other users.

- Views cannot always be meaningfully updated. A view is *updatable* if every record in the view has a unique corresponding record in some underlying database table. Single-table views that do not involve aggregation are the most natural updatable views, because there is an obvious connection between the view records and the stored records.

- Instead of allowing view updates, a DBA often writes *stored procedures.* A stored procedure contains a series of database commands embedded in a conventional programming language. A user updates the database by executing the appropriate stored procedure.

4.7 Suggested Reading

Relational algebra is defined in nearly every introductory database text, although each text tends to have its own syntax. A detailed presentation of relational algebra and its expressive power can be found in [Atzeni and DeAntonellis 1992]. That book also introduces *relational calculus*, which is a query language based on predicate logic. The interesting thing about relational calculus is that it can be extended to allow recursive queries (that is, queries in which the output table is also mentioned in the query definition). Recursive relational calculus is called *datalog*, and is related to the Prolog programming language. A discussion of datalog and its expressive power also appears in [Atzeni and DeAntonellis 1992].

There are numerous books that discuss the relational model and SQL in great detail, for example [Date and Darwen 2004]. Online SQL tutorials also exist; see, for example, *www.sqlcourse.com*. The SQL language has evolved over the years, and is now replete with features, many of which go far beyond the ideas of this chapter. (For example, the reference guide [Gulutzan and Pelzer 1999] is over 1,000 pages long.) Even with all these features, simple-sounding queries can often be difficult to express in SQL. There is consequently a market for books such as [Celko 1999] and [Bowman 2001] that give techniques for getting around SQL's limitations. A thoughtful critique of the problems with SQL appears in Chapter 14 of [Date 1986].

4.8 Exercises

CONCEPTUAL EXERCISES

4.1 Show that the *rename* operator can be simulated by using the *extend* and *project* operators. In particular, give an equivalent query to Q9 that doesn't use *rename*.

4.2 The *join* operator was defined as a special case of a selection of a product. Show that conversely, the *product* operator can be treated as a special case of join. Give an equivalent query to Q22 that doesn't use *product*.

4.3 Show that the *outerjoin* operator can be simulated by using the *join, extend, antijoin*, and *union* operators. In particular, give an equivalent query to Q55 that does not use *outerjoin*.

4.4 Show that the *antijoin* operator can be simulated as a projection and selection of an outer join. In particular give equivalent queries to Q41, Q43, Q47, and Q50 that do not use *antijoin*.

4.5 Query Q22 performed the product of STUDENT and DEPT. What is the output of that query if either one of the input tables is empty?

4.6 Figure 4-10 depicts a query tree for a circular join of three tables. Give two other equivalent query trees that join different combinations of the three tables together.

4.7 Give several other query trees that are equivalent to the tree of Figure 4-9.

4.8 Queries Q32-Q34 determine the students having the same major as Joe.
a) The output table includes Joe. Revise the queries so that it doesn't.
b) The queries assume that there is a student named Joe in the database. What does the output table look like if no student is named Joe?
c) The queries also assume that there is only one student named Joe. What does the output table look like if there are two students named Joe, each having a different major?

4.9 Consider queries Q32-Q34.
a) Write a single SQL query that is equivalent to them.
b) Revise the query as in Exercise 4.8(a).

4.10 Suppose that table T_1 contains field A and table T_2 contains field B, and consider the following relational algebra query:

```
semijoin(T1, T2, A<>B)
```

a) Describe the output table of this query.
b) Explain why this query is not equivalent to the query `antijoin(T1, T2, A=B)`.

4.11 Consider the following query:

```
join(STUDENT, DEPT, MajorId=DId)
```

Suppose that the tables are as shown in Figure 1-1, except that the records for Joe and Amy in the STUDENT table have a null value for *MajorId*.
a) What is the output table of the query?
b) Suppose that the query is an outer join instead of a join. What is the output table of the query?

4.12 In an SQL query, the order in which the tables are mentioned in the *from* clause is irrelevant. Explain why.

4.13 Consider an SQL query such as Q84. How do you suppose that SQL knows it is a *groupby* query if it doesn't have a *group by* clause?

4.14 Query Q84b is not correct as written, but it is easy to fix. Revise it using a nested query.

4.15 In Section 4.3.6 we saw that query Q87 is not correct, because it does not perform an outer join. Furthermore, the section stated that it isn't sufficient to just change the join to an outer join, because the field *HowMany* would contain nulls instead of 0's. Create a correct version of Q87 by using an outer join and an SQL case expression (or *if* function) to turn the nulls into 0's.

4.16 Revise query Q98 so that it doesn't use nested queries.

4.17 Write the definition for the view STUDENT_INFO, as discussed in Section 4.5.2.

4.18 Give an example showing why union views should not be updateable.

4.19 Are semijoin and antijoin views updateable? Explain your reasoning.

4.20 Write a query to retrieve the following information. Express your query in relational algebra, as a query tree, and in SQL. (For SQL, feel free to use views to break up the query into smaller pieces.)
a) The number of students in each section.
b) The average section size per year taught. (Note: *Avg* is an aggregation function.)
c) The sections that were larger than the average section size for their year.
d) The sections larger than 90% of the sections taught that year. (Hint: First calculate the number of sections taught each year. Then use a self-join to calculate, for each section, the number of sections smaller than it during its year. Then combine these results.)
e) The year having the largest average section size.
f) For each year, the department having the lowest average section size.
g) The (professor, course) pairs such that the professor taught that course more than 10 times.
h) The professors who have taught courses over a range of at least 20 years. (That is, the latest course taught comes at least 20 years after the earliest course taught.)
i) The (professor, course) pairs such that the professor taught that course for at least two consecutive years.
j) The professors who taught math courses for at least two consecutive years.
k) The courses that Joe and Amy have both taken.
l) The sections that Joe and Amy took together.
m) The STUDENT record of the student(s) graduating in 2005 who took the most courses.
n) The drama majors who never took a math course.
o) The percentage of drama majors who never took a math course.
p) The STUDENT record of the students who never received a grade below an “A”.
q) The STUDENT record of the students who never received a grade of “A” in any course they took in their major.

4.21 Assume that in addition to the tables of Figure 1-1, the database also contains a table GRADEPOINTS, which gives the point value of each letter grade. The table looks like this:

GRADEPOINTS	Grade	Points
	A	4.0
	A-	3.7
	B+	3.3
	…	…
	F	0.0

Write a query to retrieve the following information. Express your query in relational algebra, as a query tree, and in SQL. (For SQL, feel free to use views to break up the query into smaller pieces.)
a) The GPA of each student. (The GPA is the average points of each grade the student received.)
b) The "major GPA" of each student, which is the GPA calculated for the courses in the student's major.
c) For each student, a record containing the student's ID, name, GPA, and major GPA.
d) The student(s) having the highest GPA.
e) The average grade points per section.
f) The section of calculus taught in 2006 having the highest average grade, and the professor who taught it.

PROJECT EXERCISES

4.22 Consider the videostore database that you designed for Exercise 3.24. Write SQL queries to perform the following tasks. Feel free to use views to break up each query into smaller pieces.
(a) List the title, year, and running time of all G-rated or PG-rated movies between 1992 and 1997 that run no longer than 90 minutes.
(b) List all information about movies released in 1998 whose title begins with the letter "J", sorted by title.
(c) Produce a query having one value, which describes the number of different directors there are.
(d) Produce a query having one value, which tells the average number of movies made per year.
(e) List the years (no duplicates) for which there is some movie rated NC-17.
(f) For each director that has made at least 18 movies, give the director's name and the number of movies made. Sort by number of movies made, descending.
(g) List, for each city having at least 7 customers with that city as an address, the name of the city and the number of customers it has.
(h) For each movie, round its running time to the closest 1/2 hour; call this value the movie's *hourblock*. For each hourblock, give the number of movies in that hourblock.
(i) We say that the *range* of an actor's career is the number of years between the actor's first movie and last movie. For each actor whose range is at least 30 years, show the actor's name, the first year s/he made a movie, the last year s/he made a movie, the total number of movies s/he has made, and the range of years (between the first movie and the last movie). Sort by actor name ascending.
(j) For each time that an actor made at least 5 movies in a single year, show the actor's name, the year, and the number of movies the actor made that year. Sort by the actor name, then year.
(k) List the title, year, and director of all movies in which both John Cusack and Joan Cusack acted together. Sort by year ascending.
(l) List the title, year, and director of all movies in which John Cusack acted but Joan Cusack did not. Sort by year ascending.

(m) List the title of those movies made in 1999 for which the only listed actors are female. Sort by title ascending.
(n) List the names of those actors who have never been in a movie made in 1970 or later.
(o) Show the year having the most number of non-R-rated movies, together with that number.

5
INTEGRITY AND SECURITY

In this chapter we examine ways in which a database system can ensure the consistency and safety of its data. We consider three topics: *assertions*, *triggers*, and *authorization*. An assertion specifies an integrity constraint, and compels the database system to reject any update that violates it. By specifying assertions, the database administrator can help ensure that the data will not get updated incorrectly. A trigger specifies an action to be taken in response to a particular update. Trigger actions can perform additional updates to the database, or can fix problems with the submitted update.

Authorization is a way of restricting how users access the data. Each query and update statement requires a certain set of privileges. A user can perform a statement only if the user has sufficient privileges to do so. Authorization keeps malicious users from corrupting the database or seeing private data that they have no business seeing.

5.1 The Need for Data Integrity and Security

The data in a database is often of great importance to an organization. Because it is so important, the data needs to have *integrity* and be *secure*. By integrity, we mean that the database should not become incorrect or inconsistent, due to an inadvertent update (such as via a typo or a misunderstanding). By security, we mean that the data should not be accessible to unauthorized people (whether they be curious existing users, or malicious non-users).

Security is an issue that applies well beyond the boundaries of a database system. In particular, operating systems have to deal with many of the same threats as database systems. Some common threats are:

- a hijacked network, in which router hardware is compromised or Ethernet packets are intercepted;
- a break-in to the computing facility, in which computers and hard drives are stolen or compromised;
- identity theft, in which a non-user is able to pose as a legitimate user;
- hacked software, in which a Trojan horse or virus is implanted into the system software.

An organization can address these threats in various ways. For example, encryption can make it difficult to steal files or transmitted data packets; physical system lockdowns and alarms can make it difficult to steal equipment; and formal software update procedures can lessen the probability of corrupted software.

In addition to these general-purpose issues, a database system can address several database-specific issues.

- It can allow data providers to specify *integrity constraints*, in order to detect an inappropriate change to the database.
- It can support *triggers*, which can be used to log selected changes to the database, or specify actions that can be performed to fix inappropriate changes.
- It can support a detailed *authorization mechanism*, in which the data provider can specify, for each user, which actions that user is allowed to perform on that data.

This chapter examines these database-specific issues.

5.2 Assertions

As we saw in Chapter 2, a table may have constraints specified for it. These constraints help ensure that the table contains good, meaningful data. For example, a key constraint ensures that the key fields really do uniquely identify a record, a referential integrity constraint ensures that each foreign key value really does correspond to an existing record, and so on. Whenever a user submits an update statement, the database system first checks the requested changes against the constraints, and performs the update only if there is no violation.

In this chapter we are interested in integrity constraints. Chapter 2 showed how to specify an integrity constraint within an SQL *create table* statement. For example in Figure 2-4, the clause *check(GradYear >= 1863)* specified the constraint that each STUDENT record must have a *GradYear* value of at least 1863.

However, a *create table* statement can only specify a fraction of the possible integrity constraints, namely those constraints that apply to the individual records in a table. In order to specify an integrity constraint in SQL that applies to an entire table (or to multiple tables), we need to use *assertions*.

> *An assertion is a predicate that must always be satisfied by the database.*

For example, the assertion of Figure 5-1 expresses the constraint that no section can have more than 30 students.

```
create assertion SmallSections
   check ( not exists select e.SectionId
                      from ENROLL e
                      group by e.SectionId
                      having count(s.EId) > 30
         )
```

Figure 5-1: The SQL specification of the integrity constraint
"No section can have more than 30 students"

The *check* clause in an assertion has the same syntax as in Chapter 2: it contains a predicate inside some parentheses. The predicate in Figure 5-1 is of the form “not exists Q” for a query Q. The operator “not exists” returns *true* if the output table of its query contains no records. And since the query finds those sections having more than 30 students, the predicate will be true exactly when no section has more than 30 students.

The predicate in an assertion can be an arbitrary SQL predicate, but most commonly it is of the form “not exists Q”. Intuitively, the query Q finds violations of the constraint (such as the existence of a large section); the predicate “not exists Q” then asserts that there are no such violations.

An SQL assertion can denote any constraint whose violation can be expressed as a query. This is a wide range of integrity constraints, and quite likely covers all of the constraints you will ever encounter in your life. In contrast, the integrity constraints that can be specified in a *create table* statement are but a small special case. For example, Figure 5-2 shows how the constraint “GradYear >= 1863” would appear as an assertion.

```
create assertion ValidGradYear
   check ( not exists select s.*
                      from STUDENT s
                      where s.GradYear < 1863
         )
```

Figure 5-2: The SQL specification of the integrity constraint
“A student’s graduation year must be at least 1863”

The previous two assertions applied to a single table. Figure 5-3 gives an example of an assertion that applies to multiple tables. This assertion specifies the constraint that a student can take a course at most once.

```
create assertion NoTakeTwice
   check ( not exists select e.StudentId, k.CourseId
                      from SECTION k, ENROLL e
                      where k.SectId = e.SectionId
                      group by e.StudentId, k.CourseId
                      having count(k.SectId) > 1
         )
```

Figure 5-3: The SQL specification of the integrity constraint
“Students can take a course at most once”

5.3 Triggers

Suppose that the university wants to monitor changes to student grades. It can do so by instructing the database system to insert a record into a table (say, GRADE_LOG) whenever anyone modifies the *Grade* field of an ENROLL record. A member of the registrar's office can then periodically examine the contents of GRADE_LOG to ensure that all changes are appropriate.

This instruction to the database is called a *trigger*.

> *A trigger specifies an action that the database system should perform whenever a particular update statement is executed.*

Figure 5-4 shows how the automatic insertion into the log table would be specified as an SQL trigger.

```
create trigger LogGradeChange after update of Grade in ENROLL
   for each row
   referencing old row as oldrow, new row as newrow
   when oldrow.Grade <> newrow.Grade
      insert into GRADE_LOG(UserName, DateChanged, EId,
                            OldGrade, NewGrade)
      values(current_user, current_date, newrow.EId,
             oldrow.Grade, newrow.Grade)
```

Figure 5-4: An SQL trigger that logs changes to student grades

An SQL trigger has several parts to it. The first line of Figure 5-4 specifies the event that will initiate the trigger – here, after an update statement that modifies the field *Grade*.

The second line specifies that the trigger action will apply to each record modified by that statement. Another possibility is to say "for each statement", in which case the action will happen once.

The third line assigns variables that will let us reference the original and new versions of the modified record. Technically these variables could be named anything, but *oldrow* and *newrow* seem simplest.

The fourth line contains the *when clause*, which specifies the condition under which the trigger should fire. If this clause is omitted, then the trigger applies to all modified rows. Here, it applies only to those rows in which the value of *Grade* actually changed.

The remaining lines give the action that should be taken. Here, the action is an insert statement. The current user, current date, ID of the modified record, and the old and new grades are inserted into the GRADE_LOG table. In general, a trigger action can consist of several update statements; these statements would be bracketed by the keywords BEGIN and END.

This discussion can be summarized as follows.

A trigger has three essential parts:

- An *event*, which is the update statement that initiates the activity;
- A *condition*, which is the predicate that determines whether the trigger will fire;
- An *action*, which is what happens if the trigger does fire.

Triggers are often called *Event-Condition-Action rules*.

Triggers are often used to safeguard the database, similarly to constraints. The difference between triggers and constraints can be thought of as follows. A constraint is rigid and draconian, summarily rejecting any update that violates it. A trigger, on the other hand, is permissive and forgiving. Instead of specifying a constraint to forbid bad updates, we can often write a trigger whose action will fix the bad update.

For example, the trigger in Figure 5-4 is concerned with keeping student grades accurate. If we were to use a constraint, we might say something like “only the professor who gave the grade can change it.” The trigger, on the other hand, allows anyone to change the grade; but it also performs an additional action to ensure that each grade change can be checked for appropriateness.

For another example, suppose that we want to make sure that the graduation year of every student is no more than 4 years in the future. We could enforce a constraint that would reject inappropriate changes to the database. Or we could write a trigger that fixes inappropriate changes. Figure 5-5 shows a trigger that deals with insertions. It checks each inserted record, and if the provided graduation year is too high, it is set to null.

```
create trigger FixBadGradYear after insert in STUDENT
   for each row
   referencing new row as newrow
   when newrow.GradYear > 4 + extract(year, current_date)
      set newrow.GradYear = null
```

Figure 5-5: An SQL trigger that replaces inappropriate graduation years with nulls

Although triggers are more powerful than constraints, they are not as easy to use, because each trigger applies to a specific update statement only. For example, the trigger in Figure 5-5 fixes a bad graduation year when a record is inserted, but does nothing to protect against a modification that creates a bad graduation year.

Ideally, triggers and constraints should work together. We would create assertions to specify what it means for the data values to be correct, and triggers to specify fixes when possible. There is just one small problem with this vision. By default, SQL checks constraints after an update operation and before the trigger, which means that a bad update will be rejected before the trigger gets the chance to fix it.

We can solve this problem by deferring constraint checking. In particular, adding the phrase "*deferrable initially deferred*" to the end of an assertion (or any other constraint) tells SQL to defer checking that constraint until the end of the transaction, which gives the trigger time to fix it. However, deferred constraint checking is a dangerous path to take. If something unexpected happens early in your transaction, you won't find out about it until much later, perhaps after it is too late to fix. You need to be very confident that your triggers will pick up the slack of any deferred constraint.

5.4 Authorization

Constraints and triggers help keep the database from becoming corrupted, by catching well-intentioned but problematic updates. We now consider how to keep malicious users from corrupting the database intentionally, and how to keep overly-curious users from looking at private data.

5.4.1 Privileges

The creator of a table usually doesn't want just anybody to be able to update it. For example, the registrar's office should be the only one allowed to insert SECTION records, and professors should be the only ones who can modify grades. Moreover, some information is private, and should be visible only to specified users. For example, perhaps only the admissions office and the dean's office are allowed to look at the contents of the STUDENT table.

In SQL, users can access the database only if they have been granted the appropriate *privileges*.

> *The creator of a table grants privileges on it to other users.*

SQL has several kinds of privilege, which include *select*, *insert*, *delete*, and *update*. The table creator uses these privileges to specify, for each kind of SQL statement, which users are allowed to perform which statements. The SQL *grant statement* assigns the privileges. For example, the statement

```
grant insert on SECTION to registrar
```

assigns an insert privilege on the SECTION table to the user "registrar". The statement

```
grant select on COURSE to public
```

assigns a select privilege on COURSE to all users. Figure 5-6 lists some reasonable grant statements for the university database.

```
grant select on STUDENT to dean, admissions
grant insert on STUDENT to admissions
grant delete on STUDENT to dean
grant update on STUDENT to dean

grant select on COURSE  to public
grant insert on COURSE  to registrar
grant delete on COURSE  to registrar
grant update on COURSE  to registrar

grant select on DEPT    to public

grant select on ENROLL  to dean, professor
grant insert on ENROLL  to registrar
grant delete on ENROLL  to registrar
grant update on ENROLL  to professor

grant select on SECTION to public
grant insert on SECTION to registrar
grant delete on SECTION to registrar
grant update on SECTION to registrar
```

Figure 5-6: Some SQL grant statements for the university database

These grant statements imply that the dean's office has the ability to see all of the tables, the admissions office can see everything except the ENROLL table, and professors can see everything except STUDENT. In addition, the admissions office is authorized to insert STUDENT records, but only the dean's office can modify or delete them.

Figure 5-6 implies that everyone is allowed to look at the DEPT table, but nobody is allowed to update it. So then how do new DEPT records get created? Don't forget about the creator of the table. The creator of the table automatically has all privileges on it, and no explicit grant statement is necessary. (After all, it doesn't really make sense for the creator to grant a privilege to himself.) If we assume that the database administrator is responsible for creating the DEPT table, then it follows that the DBA is the only user allowed to update it. If the university adds (or cancels) a department, the DBA will make the necessary changes to the database.

5.4.2 Users and Roles

Some of the privileges in Figure 5-6 are granted to someone named *professor*. In Chapter 4 we had assumed that each professor logs into the database individually; that is, nobody logs in as "professor". So it seems that the grant statement

```
grant update on ENROLL to professor
```

is wrong, and should be replaced by

```
        grant update on ENROLL to einstein, newton, brando, ...
```

where we list the name of every single professor. What's the deal here?

Our proposed replacement grant statement may be correct, but it is totally infeasible. Suppose that the university has hundreds of professors; each professor would have to be listed in each professor-related grant statement, and revising the grant statements each time a professor was hired or fired would be a nightmare.

SQL avoids this difficulty by distinguishing between *users* and *roles*.

> *A user is the person who is logged into the database.*
> *A role is a category of users.*

A privilege can be granted to a user individually, or to a role. If a privilege is granted to a role, then all users belonging to that role acquire that privilege.

In a large database environment (such as a university), it often useful to create a role for each job function, and to grant privileges to the roles. Users then inherit their privileges according to their assigned roles. For example in Figure 5-6, the grantees "dean", "admissions", "registrar", and "professor" are all roles. Each professor-user is assigned to the "professor" role, each person working in the dean's office is assigned to the "dean" role, and so on. A user having more than one job function will be assigned to multiple roles. For example, if professor Brando is currently helping out in the dean's office, then he will be assigned the "dean" role in addition to his "professor" role. In this case, his privileges will be the union of the privileges of both his roles.

5.4.3 Column privileges

The purpose of granting privileges is to specify what users can and cannot do. For example, consider the following grant statements from Figure 5-6:

```
            grant select on STUDENT to dean, admissions
            grant update on STUDENT to dean
```

The first statement specifies that users in the dean's or admissions offices are allowed to view the entire table. Users without this privilege will not be able to see any part of the STUDENT table[†]. Similarly, the second statement specifies that users in the dean's office are allowed to modify any field of any record in the table.

This "all or nothing" aspect to privileges makes it difficult to assign privileges accurately. For example in Figure 5-6, only professors and the dean's office are allowed to see the ENROLL table, primarily to keep student enrollments and grades private. But this means that all of the other information in ENROLL must be equally private.

[†] In fact, the database system will not even admit to them that the table exists!

To avoid this kind of conundrum, SQL allows privileges to be granted on specific columns of a table. For example, the following statements keep the values of *StudentId* and *Grade* private, but make *EId* and *SectionId* public:

```
grant select(StudentId,Grade) on ENROLL to dean, professor
grant select(EId,SectionId)   on ENROLL to public
```

Column privileges can also be specified for update and insert operations. For example, consider the following two modifications to Figure 5-6:

```
grant update(Grade)          on ENROLL  to professor
grant insert(SName, MajorId) on STUDENT to admissions
```

The first grant statement above allows professors to change only the *Grade* values of ENROLL records, which is far more appropriate than allowing them to update all of the fields. The second grant statement allows the admissions office to assign only the name and major of a new STUDENT record; the system will assign default values for the other two fields.

5.4.4 Statement Authorization

Each SQL statement requires certain privileges to execute.

> *A user is authorized to execute a statement if that user has all of the privileges required by that statement.*

The privileges required by an SQL statement are defined by the following rules:

- A *select* statement requires a *select* privilege on every field mentioned in the query.
- An *insert* (or *update*) statement requires an *insert* (or *update*) privilege for the fields to be inserted (or modified), plus a *select* privilege on every field mentioned in its *where* clause.
- A *delete* statement requires a *delete* privilege for the table, plus a *select* privilege on every field mentioned in its *where* clause.

These rules are based on the following intuition. Since it is possible to determine the result of a query by looking only at the values of its mentioned fields, the user should need *select* privileges only on those fields. Similarly, since the *where* clause of an update statement determines which records will be updated, the user should need *select* privileges on those fields, as well as the appropriate update privilege.

Statement	Required Privileges
select s.SId, s.SName, count(e.EId) from STUDENT s, ENROLL e where s.SId = e.StudentId and e.Grade = ‘A’ group by s.SId, s.SName	select(SId, SName) on STUDENT select(EId, StudentId, Grade) on ENROLL
select c.* from COURSE c where c.DeptId in (select d.DId from DEPT d where d.DName = ‘math’)	select on COURSE select(DId, DName) on DEPT
delete from SECTION where SectId not in (select e.SectionId from ENROLL e)	select(SectId) on SECTION delete on SECTION select(SectionId) on ENROLL

Figure 5-7: Some SQL statements and their required privileges

For example, Figure 5-7 gives some example SQL statements, together with their required privileges. By using the privileges granted in Figure 5-6, we can infer the following authorizations:

- The first statement requires *select* privileges on STUDENT and ENROLL. There are two kinds of user who have these privileges: users in the dean’s office, and users in the admissions office who happen to also be professors.
- The second statement requires *select* privileges on COURSE and DEPT. Since both of these privileges are granted to “public”, every user is authorized to execute this statement.
- The third statement requires *select* privileges on SECTION and ENROLL, as well as *delete* privilege on SECTION. The only users authorized to execute this statement must be in the registrar’s office as well as either being a professor or in the dean’s office.

The user authorization for the third statement seems overly restrictive. This statement ought to be executable by anyone in the registrar’s office, in order to delete sections that have no enrollments. The problem is that the registrar’s office does not have *select* privilege on ENROLL, because some fields (such as *Grade*) are private. If we had column privileges, however, things would be different. In particular, suppose that the field *SectionId* of ENROLL is granted to public, as discussed earlier; in this case, the registrar’s office will be authorized for the statement.

5.4.5 Privileges on views

Column privileges are useful because they fine-tune the power granted to each user. For example, consider what privileges a professor ought to have on the ENROLL table. If professors have *update* privilege on all of ENROLL, then they are able to update grades (which is good), but they are also able to change section enrollments (which is bad). We can limit their power by granting them column privileges instead – namely, *update* privilege on the *Grade* field of ENROLL.

However, this column privilege still gives them too much power, because it allows a professor to change the grade given by another professor. What we really want is a privilege that only allows professors to change the grades of their own students. Such privileges do not exist in SQL, but we can simulate them by using views. Consider the SQL statements in Figure 5-8. The view PROF_ENROLLMENTS returns the subset of ENROLL records corresponding to the logged-in professor. The grant statements allow professors to see all of these records, and to update their grades.

```
create view PROF_ENROLLMENTS as
      select e.*
      from ENROLL e
      where e.SectionId in (select k.SectId
                            from Section k
                            where k.Prof = current_user)

grant select        on PROF_ENROLLMENTS to professor
grant update(Grade) on PROF_ENROLLMENTS to professor
```

Figure 5-8: Using a view to limit the power of professors

Suppose that we revoke the privileges that were granted to professors on the ENROLL table, and instead grant them the privileges of Figure 5-8 on the view PROF_ENROLLMENTS. Professors will still be allowed to browse the public columns of the ENROLL table (because public privileges apply to everyone), but any professor-specific activity must be performed through the view. Note that the view is updatable, because of its single-table outer query; thus updating the grade makes sense.

The privileges on the view can be thought of as *customized privileges*, because we created the view for the sole purpose of granting privileges on it to some user. There is a close correspondence between this situation and the way the database administrator creates an external schema: In both cases, the views are created specifically for some user, and that user is given privileges only on the views.

5.4.6 Privileges on constraints

Let's now consider assertions. Recall that the body of an assertion is a predicate that can possibly refer to multiple tables. Should there be any restrictions on who can create an assertion?

The answer is an emphatic "yes". Creating an assertion restricts the contents of the mentioned tables, and therefore limits what users can do. It would be very easy for a malicious user to create an assertion that would render a table useless. A simple example is an assertion on STUDENT that required that the *SName* value of each record be “xyz”, the *GradYear* value be 0, and the *MajorId* value be null.

Clearly, writing assertions (as well as other constraints) is serious business. If you allow another user to write an assertion that restricts your tables, you had better trust that user a lot. In SQL, the creator of a table expresses this trust by granting a *references* privilege to that other user. In other words, the grant statement

```
grant references on STUDENT to dean
```

specifies that the dean’s office is allowed to create constraints that mention the STUDENT table.

5.4.7 Grant-option privileges

Finally, we come to the grant statement. We know that the creator of a table not only has all privileges on the table, but is also able to grant these privileges to others. The creator has one additional ability – The creator can grant a privilege to grant a privilege on the table. Such a privilege is called a *grant-option privilege*.

Each regular privilege has a corresponding grant-option privilege. Grant-option privileges are assigned using a variation of the grant statement. For example, the following statement grants two privileges to the admissions office: the privilege to insert records into STUDENT, and the privilege to grant that privilege to others.

```
grant insert on STUDENT to admissions with grant option
```

By assigning a grant-option privilege to another user, the grantor is delegating privilege-granting authority to that user. For example, suppose that the above grant statement was executed by the database administrator. Presumably, the DBA feels that since the admissions office knows how the university processes new students, it also knows best about how new STUDENT records get created; the DBA therefore trusts the admissions office to assign the insert privileges appropriately.

The grant-option privilege is actually very powerful. The admissions office now not only has the privilege to grant insert privileges on STUDENT to others, it also has the privilege to grant the grant-option privilege. For example, it could decide that the registrar is equally qualified to grant insert privileges on STUDENT, and thus gives the grant-option privilege to the registrar. At this point, the DBA, the admissions office, and the registrar would all have the grant-option privilege. The issue at hand is (again) one of trust. The creator of a table needs to be very careful about who gets grant-option privileges on it, because those users can then propagate the grant-option privilege to others, without the creator’s knowledge or approval.

5.5 Mandatory Access Control

Let’s assume that the university is serious about keeping student grades confidential. It therefore grants SELECT privilege on the ENROLL table only to the users who need to see them, namely professors and members of the dean’s office.

There is no question that these privileges ensure the confidentiality of the ENROLL table. Unfortunately, they do not ensure the confidentiality of the grades. The problem is that a disreputable authorized user (say, a professor) can easily violate confidentiality. One simple approach is for the professor to execute the SQL statements of Figure 5-9.

```
create table TakeAPeek(StudentId int, Title varchar(20),
                       Grade varchar(2));

grant select on TakeAPeek to public;

insert into table TakeAPeek(StudentId, Title, Grade)
      select e.StudentId, c.Title, e.Grade
      from ENROLL e, SECTION k, COURSE c
      where e.SectionId=k.SectId and k.CourseId=c.CId;
```

Figure 5-9: An easy way to abuse authorization

That is, the professor can create a table and insert the confidential grade information into it. Now since the professor is the creator of the table, the professor has grant-option privileges on it, and can therefore authorize anybody to look at it.

The SQL authorization mechanism is called *discretionary access control*, because the creator of a table is able to authorize access to it, at his discretion. A database system that uses discretionary access control depends on its users to be trustworthy. So if a user has not been given grant-option privilege on a table, then the database system trusts that the user will not exploit loopholes (as in Figure 5-9) that simulate that privilege.

A better way to ensure confidentiality is to assign privileges to data, instead of tables. That way, if a user inserts confidential data from the ENROLL table into another table, then that data will remain confidential. This mechanism is called *mandatory access control*, because the privileges on a table are determined automatically by the privileges associated with its contents. The creator of the table has no choice in the matter.

Privileges in discretionary access control are associated with tables.
Privileges in mandatory access control are associated with data.

So how does one associate a privilege with data? The most common approach is what is used by the military. Every record in the database is assigned a *classification level*, typically by the user who performs the insertion. The database supports a fixed set of classification levels. For example, the U.S. Military uses four levels: *Top Secret*, *Secret*, *Classified*, and *Unclassified*. These levels form a strict order, with *Top Secret* having the highest level and *Unclassified* having the lowest level.

In general, it is possible for the records in a table to have different classification levels. However, doing so adds significant complexity, and is beyond the scope of this chapter.

Instead, we shall assume here that all of the records in a table have the same level. We define the *level of a table* to be the level of the records in it.

Each user is also assigned a classification level. A user is authorized to execute an SQL statement if the user's level is at least as high as the level of the input tables. For example, suppose that ENROLL has the level *Secret*, SECTION has the level *Classified*, and COURSE has the level *Unclassified*. Then a user would be able to execute the query of Figure 5-9 only if that user was assigned the level *Secret* or *Top Secret*.

The level assigned to the output of a query is the highest level of its input tables. So for example, the output of the query from Figure 5-9 would all be assigned the level *Secret*. Note that this rule applies even if all of the output fields come from an unclassified table. For example, consider the query of Figure 5-10, which returns the titles of courses in which somebody received an "F". Since course titles are unclassified, it seems a bit strange to say that a particular subset of course titles must be treated as secret. However, we simply must treat them as secret, because we used secret information to determine them.[†]

```
select c.Title
from ENROLL e, SECTION k, COURSE c
where e.SectionId=k.SectId and k.CourseId=c.CId
and e.Grade = 'F'.
```

Figure 5-10: What should be the classification of this query's output table?

The purpose of mandatory access control is to improve the security of a database system by removing some of the loopholes. However, commercial databases rarely use this mechanism, for two principal reasons.

The first reason is that discretionary access control allows for a much wider range of authorization policies. In mandatory access control there is small set of classification levels, and all users at a particular level are able to see all data at that level or below. There is no way to customize access for specialized groups of users, as there is with discretionary control.

The second reason is that mandatory access control closes only the database-related loopholes. If an organization needs that high of a level of security, then it also will need to be much more vigilant about closing other loopholes. For example, even under mandatory control, a renegade professor could give confidential grade information to people who call him over the phone, or he could publish his username and password on the internet, or he could leave his office with the door unlocked and his computer connected to the database. And of course, there are all of the hacker-related loopholes

[†] Here is perhaps a more dramatic illustration of this point: A list of employees might be unclassified, but the list of employees suspected of being foreign spies is not.

mentioned at the beginning of the chapter. If mandatory access control is to be effective, then these other loopholes must also be addressed.

Consequently, most organizations find mandatory access control to be ill-suited for their needs. Instead, discretionary control, coupled with detailed audit trails and managerial oversight, seem to be sufficient.

5.6 Chapter Summary

- An assertion is a predicate that must always be satisfied by the database. Typically, the predicate in an assertion is of the form "not exists Q". Intuitively, the query Q finds violations of the constraint (such as the existence of a large section); the predicate "not exists Q" then asserts that there are no such violations.

- A trigger specifies an action that the database system should perform whenever a particular update statement is executed. A trigger has three parts:
    - An *event*, which is the update statement that initiates the activity;
    - A *condition*, which is the predicate that determines whether the trigger will fire;
    - An *action*, which is what happens if the trigger does fire.

- Assertions and triggers can be used to catch well-intentioned but problematic updates. There are two primary differences: An assertion is enforced regardless of how the database gets changed, whereas a trigger applies only to a specified update statement. One the other hand, triggers can respond to bad updates in a variety of ways, whereas an assertion responds by rejecting the update.

- Privileges are used to restrict a user's access to the database. A user is authorized to execute a statement only if that user has all of the privileges required by that statement.

- The privileges required by an SQL statement are defined by the following rules:
    - A *select* statement requires a *select* privilege on every field mentioned in the query.
    - An *insert* (or *update*) statement requires an *insert* (or *update*) privilege for the fields to be inserted (or modified), plus a *select* privilege on every field mentioned in its *where* clause.
    - A *delete* statement requires a *delete* privilege for the table, plus a *select* privilege on every field mentioned in its *where* clause.

- The creator of a table has all privileges on it. The creator can assign these privileges to other users by issuing grant statements.

- A privilege can be granted to a user individually, or to a role. If a privilege is granted to a role, then all users belonging to that role acquire that privilege.

- Views can be used to fine-tune user access. The idea is to create a view containing exactly what you want a user to see, and then to grant privileges on that view. This technique is the same as what the database administrator does when creating an external schema.

- SQL authorization is an example of *discretionary access control*, because the creator of a table has the discretion to grant privileges on it to anybody.

- Another option is to use *mandatory access control*. Here, each record is assigned a classification level. A user is authorized to see the record if the user's classification level is at least as high as that of the record.

- Mandatory access control eliminates some of the security loopholes of discretionary control, but it allows for less flexible policies. It is therefore suited primarily for high-security systems, such as in the military.

5.7 Suggested Reading

There are many other uses for triggers, apart from their connection to constraints. Their ability to respond to events allows databases to be an *active* part of a software system, instead of a passive data repository. For details, see [Ceri et al 2000] and [Widom and Ceri, 1996]. However, triggers have a subtle and tricky semantics. If triggers are used indiscriminately, then a single update can spawn a chain of updates whose result may be difficult to determine (and in fact may be an infinite loop). Triggers require a certain expertise to use correctly.

The book [Castano et al 1995] contains a comprehensive coverage of database security, and includes numerous variations of both discretionary and mandatory access controls. The paper [Stahlberg et al 2007] explains why current database systems do a bad job of handling confidential data, and shows how a person can use forensic analysis to discover data that he ought not to see.

5.8 Exercises

CONCEPTUAL EXERCISES

5.1 An assertion can express a key constraint.
a) Write an assertion that *SId* is a key of STUDENT.
b) Write an assertion that {*StudentId*, *SectionId*} is a key of ENROLL.

5.2 Consider the foreign key *StudentId* in ENROLL. Write an assertion equivalent to its foreign key constraint.

5.3 Consider the constraint that says that the field *SName* in STUDENT cannot be null. Write an assertion equivalent to this constraint.

5.4 The database system needs to validate each assertion prior to every update statement. Clearly, not all assertions need to be checked for all updates; for example, the assertion *SmallSections* in Figure 5-1 cannot be violated when a new student is inserted. Develop an algorithm that identifies, given an update statement and an assertion, whether that update could possibly violate that assertion.

5.5 Revise the assertion *NoTakeTwice* so that students can take a course several times, but can only receive a passing grade once.

5.6 Write assertions for the following integrity constraints, using the tables of Figure 1-1.
a) A professor always teaches the same one course. (That is, the FD `Prof→CourseId` holds.)
b) Only math majors can take the course titled 'Topology'.
c) Students cannot take the course 'Calculus 2' without having received a passing grade in 'Calculus 1'.
d) A student can take no more than 10 courses in any year, and no more than 5 courses in any one department per year.
e) All of the courses taught by a professor are offered by the same department.

5.7 The trigger of Figure 5-5 makes it impossible for a user to insert a STUDENT record having an inappropriately-high graduation year. However, it does not stop a user from modifying the graduation year of an existing record to be too high. Write another trigger that fixes inappropriate modifications.

5.8 Suppose that we want to ensure that the *SName* values of STUDENT records are never modified.
a) One way to do this is to write a trigger that intercepts a modification to the field and sets the value back to what it was. Write this trigger.
b) Another way is to specify appropriate privileges. Give the appropriate grant statements and explain why this way is better.

5.9 Consider a SECTION record that has exactly one ENROLL record connected to it.
a) Write a trigger to delete that SECTION record when that ENROLL record is deleted.
b) How does this trigger differ from a constraint that says that a section must have at least one student?

5.10 Revise the grant statements in Figure 5-6 to use column privileges. Defend your decisions.

5.11 The creator of the STUDENT table needs a REFERENCES privilege on the DEPT table in order to specify that *MajorId* is a foreign key. Explain what malicious things the creator could do if that privilege were not necessary.

5.12 SQL does not let a user create a view unless that user is authorized to execute the view definition.
a) Explain why this is an important restriction.

b) Under what conditions do you think the view creator should get *insert*, *update*, or *delete* privileges on the view?

5.13 Write a view that allows students to see their grades but nobody else's. Give the SQL statements to grant the necessary privileges on the view.

5.14 Consider a multi-user operating system that you are familiar with.
a) How does it implement authorization?
b) How does OS authorization compare to SQL authorization?

5.15 Investigate a high-security database site that uses mandatory access control. What are all of the ways that they use to keep their data confidential?

PROJECT EXERCISES

5.16 Consider the videostore database that you designed for Exercise 3.24.
a) Determine some appropriate multi-table integrity constraints, and implement them as assertions.
b) Write a trigger that inserts a record into a log table each time a sale occurs.
c) Write triggers to enforce one of the constraints from part (a).
d) Develop an appropriate set of roles, and issue grant statements for them.

6

IMPROVING QUERY EFFICIENCY

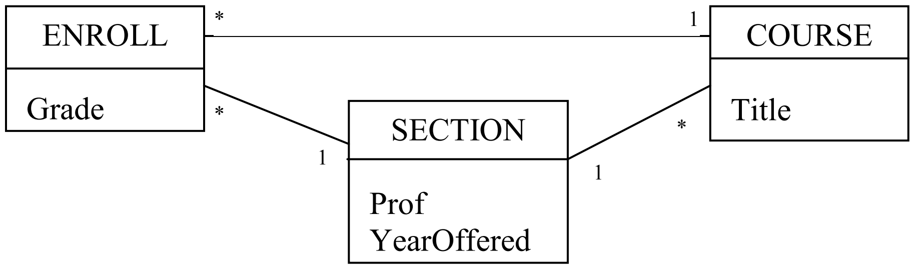
Modern database systems are able to execute SQL statements reasonably efficiently. For large databases, however, “reasonably efficient” execution can be unacceptably slow. In such cases, the database designer can specify additional constructs that will increase the speed of query execution.

This chapter considers two such constructs: *materialized views*, and *indexes*. A materialized view is a table that holds the output of a pre-computed query. An index is a file that provides quick access to records when the value of a particular field is known. A database system can use them to speed up query execution, often by a large amount. However, these constructs entail certain costs – they require additional space, and need to be maintained when the database is updated. Therefore, we also examine how a designer can determine the most cost-effective set of materialized views and indexes.

6.1 The Virtues of Controlled Redundancy

Chapter 3 stressed that a well-designed database should not contain redundant data. Indeed, there is no question that redundancy is bad, because it opens the door for inconsistent updates, and inconsistent updates lead to a corrupted database. Moreover, redundancy is inelegant. Eliminating redundancy from a class diagram almost always results in a better design.

However, redundancy also has its good side – namely, it allows users to pose more efficient queries. For example, consider the class diagram in Figure 6-1(a). This diagram is similar to our diagram of the university database; however, it has a redundant relationship between ENROLL and COURSE. Transforming this diagram results in the relational schema of Figure 6-1(b).



(a) A class diagram with a redundant relationship

```
ENROLL(EId, SectionId, CourseId, Grade)
SECTION(SectId, CourseId, Prof, YearOffered)
COURSE(CId, Title)
```

(b) The corresponding relational schema

Figure 6-1: The effect of a redundant relationship

Note how the redundant relationship causes the ENROLL table to contain the redundant foreign key, *CourseId*. Now suppose we want to know the titles of the courses in which grades of 'F' were given. The SQL query of Figure 6-2(a) takes advantage of this redundant field.

```
select c.Title
from COURSE c, ENROLL e
where c.CId=e.CourseId and e.Grade='F'
```

(a) Using the redundant field

```
select c.Title
from COURSE c, SECTION k, ENROLL e
where c.CId=k.CourseId and k.SectId=e.SectionId
and e.Grade='F'
```

(b) Without using the redundant field

Figure 6-2: The value of a redundant *CourseId* field

The redundant *CourseId* field makes it possible to join COURSE and ENROLL directly. If this redundant field did not exist, then our query would have had to include SECTION as well, as shown in Figure 6-2(b). In other words, the redundant foreign key has saved us the cost of doing a join. In a large database, this savings could be substantial.

Although we love the cost savings that the redundant foreign key brings us, the possibility of an inconsistent update is a huge liability. For example, suppose we modify the *CourseId* value of an ENROLL record, but forget to modify its related SECTION record (or vice versa). The queries in Figure 6-2 would then give us different answers. Such a situation would be unacceptable, because no amount of efficiency can compensate for incorrect results.

The above problem occurred because a user was responsible for ensuring that the redundant data is consistent, and users sometimes forget. The solution, therefore, is to put the database system in charge of data consistency. If we could tell the database system that a field is redundant, then it could make sure that a change to one record is reflected in the corresponding other records. Such a situation is called *controlled redundancy*. Because the database system understands where the redundancy is, it is able to control it and ensure that the data remains consistent.

> *Controlled redundancy is useful, because the database system can ensure that the database remains consistent.*

The remainder of this chapter considers two forms of controlled redundancy: materialized views, and indexes.

6.2 Materialized Views

6.2.1 Creating a materialized view

Section 4.5 introduced us to views. As we saw in that section, a view is essentially a named query. A user can treat a view as if it were a stored table, by mentioning it in queries and by performing updates on it (if the view is updatable). A view differs from a stored table in that its contents will automatically change when the contents of its underlying tables change.

Suppose that a user executes a query that mentions a view. The query processor can handle the view in two ways:

- It can execute the view definition as part of the query, computing its records as needed. This approach is called *query modification*, and is discussed in Chapter 19.
- It can execute the view definition first, saving its output in a temporary table. The query then references that table. This is the approach that we are interested in here.

The nice thing about the temporary-table approach is that the temporary table can be reused. If another query comes along that mentions the view, then the database system can simply use the temporary table, and save itself the expense of recomputing the view.

If the database system keeps this temporary table around permanently, we call the table a *materialized view*.

A materialized view is a table that contains the output of a view.
The table is automatically updated whenever its underlying tables change.

There are nontrivial costs involved with keeping a materialized view. One cost is space-related: The table that holds the materialized view needs to be saved on disk somewhere; and the larger the view, the more significant the impact. The other cost is time-related: Each time that the underlying tables are updated, the database system must also update the materialized table so that it is consistent with those updates. Depending on the frequency of database updates, the time required to update the table could be significant. We say that the materialized view needs to be *maintained*.

A view is worth materializing when its benefits outweigh its costs.

The benefit of keeping the view is high when the view is accessed frequently and its output is expensive to recompute each time. The cost is high when the view is updated frequently and its output is large. Since the database system doesn't have a clue about the frequency of user accesses and updates, it cannot reasonably decide whether a view should be materialized.

However, the database designer *is* able to determine these frequencies. By analyzing how the database will be used, the designer can weigh the costs and benefits, and determine which views are worth materializing. Consequently, the more sophisticated database systems (such as Oracle, DB2, and SQL Server, but *not* the SQL standard) let the designer specify that a view should be materialized. The syntax differs among different vendors; here we shall use Oracle's syntax[†].

As an example, suppose that we want our ENROLL table to contain the redundant foreign key *CourseId*. Instead of using a redundant relationship (as in Figure 6-1), we can create the materialized view of Figure 6-3. Note that the statement to create a materialized view is just a regular view definition with the added keyword *materialized*. (The general form of the statement, which we shall ignore, also gives the creator the option of specifying certain parameter values, such as how frequently the system should update the materialized table).

```
define materialized view ENROLL_PLUS_CID as
    select e.*, k.CourseId
    from ENROLL e, SECTION k
    where e.SectionId = k.SectId
```

Figure 6-3: A materialized view that adds *CourseId* to ENROLL

[†] Not only cannot the vendors agree on syntax, they cannot even agree on terminology. Oracle uses our term *materialized view*, whereas DB2 uses *materialized query table*, and SQL Server uses *indexed view*.

You should convince yourself that the view ENROLL_PLUS_CID will contain the same records as the modified ENROLL table of Figure 6-1(b). The difference is in how the redundant *CourseId* values are kept consistent. In the modified ENROLL table, the user was responsible for ensuring their consistency, whereas in ENROLL_PLUS_CID, consistency is handled by the database system.

6.2.2 Maintaining a materialized view

So how does the database system can ensure the consistency of a materialized view in the face of changes to the underlying tables?

We'll start by ruling out the most obvious approach, which is for the system to first delete the existing version of the materialized view, and then recompute it from scratch using the updated tables. This approach is much too time-consuming to be practical. Instead, the system needs to be able to update the materialized table incrementally.

The database system makes incremental changes to a materialized view each time one of its underlying tables changes.

Incremental update is best understood in terms of relational algebra, because each operator has its own update algorithm. Some operators, such as *select* and *join*, are straightforward. Other operators, such as *groupby* and *outerjoin*, require a more intricate algorithm. We shall consider the *join* and *groupby* operators here; the remaining operators are left for the exercises.

Consider the materialized view ENROLL_PLUS_CID from Figure 6-3. This view is essentially a join between the ENROLL and SECTION tables. Suppose that a user makes a change to the ENROLL table.

- If a record *r* is inserted into ENROLL, then the system joins *r* with the SECTION table, and inserts the output records (if any) into the view table.
- If a record *r* is deleted from ENROLL, then the system deletes all records from the view table that contain the specified *r*-values.
- If a field of a record *r* is modified, then there are two cases. If that field is not part of the join predicate, then the system modifies all records from the view table that contain the specified *r*-values. However, if that field is part of the join predicate, then several records could be affected. The best option is to treat the modification as if the original version of *r* is deleted, and then the modified version of *r* is inserted.

Changes to SECTION are handled analogously.

Figure 6-4 gives a materialized view that corresponds to a *groupby* operator. We'll consider how the insertion, deletion, and modification of a STUDENT record affects this view.

```
define materialized view STUDENT_STATS as
   select MajorId, count(SId) as HowMany,
          min(GradYear) as MinGradYear
   from STUDENT
   group by MajorId
```

Figure 6-4: A materialized view defined by aggregation

Suppose that we insert a new record into STUDENT. The STUDENT_STATS view will be updated differently, depending on whether the view already contains a record for that *MajorId* value. If so, then the system needs to increment the value for *HowMany*, and if the new *GradYear* value is small enough, change the value of *MinGradYear*. If not, then a new record is inserted into the view; this record will have a value of 1 for *HowMany*, and the *GradYear*-value for *MinGradYear*.

Now suppose that we delete a record from STUDENT. The system will decrement the value for *HowMany* in the corresponding view record; if the value goes to 0, then the view record will be deleted. Dealing with the *MinGradYear* value is trickier. The system needs to keep track of how many STUDENT records have the minimum value for *GradYear*. If the deleted record is not one of those records, then nothing need occur. Otherwise, the count needs to be decremented. And if the count goes to 0, then the database system will need to requery the STUDENT table to calculate a new minimum graduation year for that major.

Finally, suppose that we modify a field of a STUDENT record. Different actions will be required, depending on which field is modified.

- Modifying *GradYear* requires that the new value be checked against the current *MinGradYear* value, updating the view (and the count of records having the minimum grad year) if necessary.
- Modifying *MajorId* is best thought of as deleting the existing record, and then inserting a new record that has the new *MajorId* value.
- Modifying any of the other fields has no effect on the view.

6.2.3 Materialized views vs. denormalization

We now consider how materialized views fit into the design process of Chapter 3. Recall that this design process produces *normalized* relational schemas. A normalized schema contains one table for each kind of entity in the database. The tables therefore tend to be small, with lots of foreign keys relating them. Normalized schemas have many benefits: Each table embodies a single well-understood concept, there is no redundancy, and no chance of inconsistency.

However, normalized schemas do have a practical problem. Almost any interesting query will involve data from multiple tables, and will therefore need to join those tables together. And these queries will run quite slowly, because joins tend to be time-consuming.

Historically, database designers have been reticent about creating normalized tables. Their position was that it doesn't make sense to store data in separate tables if those tables are going to be joined together anyway. Thus they would look at a normalized relational schema, figure out which tables will typically get joined together, and create a database consisting of one or more of those (large) joined tables. This technique is called *denormalization*.

> *Denormalization is the technique of selectively joining the tables in a normalized schema in order to produce larger, unnormalized tables, solely for the sake of efficiency.*

For example, let's consider again the university database, whose normalized schema appears in Figure 6-5(a). Suppose that we determine that queries involving students will almost always request their grades, and that queries involving sections will almost always request the course title. Then we could denormalize the database by combining STUDENT and ENROLL into a single table, as well as combining SECTION and COURSE into a single table. The resulting schema appears in Figure 6-5(b). Queries on this schema will almost certainly be more efficient than queries on the original schema.

```
STUDENT(SId, SName, GradYear, MajorId)
DEPT(DId, DName)
ENROLL(EId, StudentId, SectionId, Grade)
SECTION(SectId, CourseId, Prof, YearOffered)
COURSE(CId, Title, DeptId)
```

(a) A normalized schema

```
STUD_ENR(EId, SId, SName, GradYear, MajorId, SectionId, Grade)
SECT_CRS(SectId, Prof, YearOffered, CId, Title, DeptId)
DEPT(DId, DName)
```

(b) A denormalized version of the schema

Figure 6-5: An example of denormalization

The problem with denormalization is that it replaces a good database design with a bad one, in the name of efficiency. You should convince yourself that the denormalized tables are not key-based, and have all of the update-related problems associated with non-key-based tables. In the early days of relational databases, these problems were accepted as the price that one pays to have an efficient database. Now, however, denormalization is not necessary, because you can get the same efficiency by simply adding materialized views to the normalized schema.

For example, we can add the materialized views of Figure 6-6 to the normalized schema of Figure 6-5(a). This database is similar to the denormalized one, in that user queries can avoid joins by mentioning the materialized views instead of the base tables.

```
create materialized view STUD_ENR as
    select s.*, e.EId, e.SectionId, e.Grade
    from STUDENT s, ENROLL e
    where s.SId= e.StudentId;

create materialized view SECT_CRS as
    select c.*, k.SectId, k.Prof, k.YearOffered
    from SECTION k, COURSE c
    where k.CourseId=c.CId
```

Figure 6-6: Materialized views that simulate denormalization

However, the use of materialized views has an added advantage over denormalization, which is that the normalized tables are also available. So for example, a user that is interested only in data from STUDENT can execute a query on that table. That query will be much more efficient than the corresponding query on STUD_ENR, because STUDENT is a much smaller table and therefore can be scanned more quickly. An even more important example involves user updates. Performing updates directly on the normalized tables eliminates the possibility of inconsistency that plagues denormalized databases.

6.2.4 Choosing the best set of materialized views

A materialized view is a redundant table that is maintained by the database system. The purpose of this redundancy is to give users an option when creating queries, allowing them to create the most efficient query that they can.

For example, suppose that our schema contains the materialized views of Figure 6-6, and assume that a user wants to know the titles of the courses taken by Joe. Either of the queries in Figure 6-7 can do the job. However, the second query involves fewer joins, and thus is the more efficient (and more desirable) query.

```
select c.Title
from STUDENT s, ENROLL e, SECTION k, COURSE c
where s.SId=e.StudentId and e.SectionId=k.SectId
and k.CourseId=c.CId and s.SName='Joe'
```

(a) Using the normalized tables

```
select kc.Title
from STD_ENR se, SECT_CRS kc
where se.SectionId=kc.SectId and se.SName='Joe'
```

(b) Using the materialized views

Figure 6-7: Two queries that find the courses taken by Joe.

Wait a minute. There is something disconcerting about letting users mention materialized views in their queries. It is certainly a good idea to use a materialized view to improve a query. However, do we really want the user to get involved? If we look back at Chapter 4, we see that the SQL language is intended to hide implementation concerns from the user. A database system has a query optimizer, whose job is to find the most efficient implementation of user queries. A user should be able to pose any SQL query whatsoever, without any concern for efficiency; the optimizer is then responsible for finding the most efficient way to implement it.

In other words, the decision to use a materialized view in a query should be made by the query optimizer, not the user. A user should not have to know which materialized views exist, or think about how to construct the most efficient query. That effort is the responsibility of the database system. Consider again the example of Figure 6-7. If a user poses the first query, the optimizer should automatically transform it into the second one. This is indeed what happens.

A database system can use materialized views to optimize an SQL query, even when that query does not mention those views.

This feature makes the user's life simpler, because the user can focus on what data to retrieve instead of how to retrieve it. The feature also simplifies the life of the database designer, because the purpose of regular views and materialized views are now quite distinct:

- *Regular views are for the convenience of the users.* The database designer specifies these views in order to provide a customized set of tables for each user, so that the users see the data in the way that best fits their needs. For example, the external level schemas of Section 1.7 should be implemented as regular views.

- *Materialized views are for efficiency.* The database designer specifies these views in order to pre-compute the joins, aggregations, and other time-consuming operations that occur frequently. These views are invisible to the users.

A materialized view, therefore, does not need to be structured in a way that is user-friendly. Its purpose is exclusively to hold redundant data that will be used to optimize queries.

In order to decide what materialized views to specify, a database designer must perform a detailed analysis of what queries are likely to be asked, what updates are likely to be made, and how frequently these actions will occur. This is not an easy task.

For example, consider again the two materialized views of Figure 6-6. Each view materializes the join of two tables. But there are many other possible view definitions. We could for example create a view that materializes the join of all four tables. This view can either replace the other two materialized views, or it could be added to them. How do we decide? A view that joins all four tables will be very useful for queries that mention those tables, but it will be useless for queries that mention fewer tables. On the other hand, the smaller, two-table views can be used for optimizing four-table joins (as in Figure 6-7), but not as efficiently as a single, large view.

In general, views having larger, complex definitions are able to optimize the larger, more complex queries, but are unusable for smaller queries. Conversely, views having smaller definitions can be used to optimize more queries, but less efficiently. The only thing a designer can do is analyze how frequently each kind of query is expected to occur, and thereby calculate the expected cost savings.

Commercial database systems often contain tools to estimate the impact of a materialized view, and tools to observe the actual performance of user queries with and without those views. These tools can help a designer choose which set of materialized views to define.

6.3 Indexing

6.3.1 Table organization

Another factor that affects query efficiency is how the records in each table are organized. For a good example of this issue, consider the white pages of a telephone book.

A telephone book is essentially a large table, whose records list the name, address and phone number of each subscriber. This table is sorted by subscriber's last name and then by first name. Suppose that we want to retrieve the phone number for a particular person. The best strategy is to use the fact that the records are sorted by name. For example if we do a binary search, we can locate the phone number by examining at most $\log_2 N$ listings, where N is the total number of listings. This is exceptionally quick. (For example, suppose that N=1,000,000. Then $\log_2 N < 20$, which means that we would never need to examine more than 20 listings.)

There are many ways to organize a collection of records, such as via sorting or hashing. Chapter 21 examines several of these organizational techniques in detail. These techniques will have different implementations and characteristics, but they all share the same important feature – They enable us to retrieve the records we want, without having to search the entire table. That is, a database designer typically doesn't care care *how* a table is organized; the designer just cares *whether* it is organized, and on what fields.

The advantage to organizing a table on some fields is that we can quickly retrieve all records having a particular value for those fields.

Although a telephone book is great for retrieving listings by subscriber name, it is not very good for other kinds of retrieval, such as finding the subscriber who has a particular phone number or lives at a particular address. The only way to get that information from the telephone book is to examine every single one of its listings. Such a search can be quite slow.

If we want an efficient way to look up subscribers given a phone number, then we need a telephone book that is sorted by phone number (otherwise known as a "reverse telephone book"). Of course, this telephone book is useful only if we know the telephone number. If we only have a reverse telephone book and want to know the telephone number of a particular subscriber, then we would again have to examine every single one of the book's listings.

The general principle of table organization is this:

A table can be organized only one way at a time.

If we want retrievals to be fast given either a phone number or a subscriber name, then we need two separate copies of the telephone book, each having a different organization. And if we also wanted fast retrieval of a phone number given an address, we would need a third copy of the telephone book, organized by address.

We can apply this principle to any database table. One way that we could efficiently execute multiple kinds of queries on a table is to keep redundant copies of the table, each organized on a different field. Unfortunately, this redundancy has all of the problems that we have seen before:

- *Excessive space usage*. Each additional copy is the same size as the original table. So keeping two copies of a table requires three times the space.
- *The extra time required for synchronization*. Whenever a user changes the table, all of its copies will also need to be changed. So keeping two copies of a table means that each update operation will take three times as long.
- *The potential for inconsistency*. If the user is responsible for synchronizing the various versions of a table, then there is a chance for error. And an out-of-synch copy is dangerous, because it will give the wrong answer to a query.

The solution to these problems is to use an *index* instead of a copy. An index is smaller than a copy, is easier to maintain, and its redundancy is controlled by the database. The remainder of this chapter examines indexes and their use.

6.3.2 Creating indexes

Suppose that we want to be able to quickly find the STUDENT records having a particular *MajorId* value. The proper thing to do is to create an index for STUDENT, organized on *MajorId*. An index for STUDENT is similar to a copy of the STUDENT table, but is smaller and less expensive to maintain.

Figure 6-8 gives the SQL statements for creating this index, as well as a second index that is organized on *SId*. The syntax used in this figure is common to many database systems, although it is not yet part of the SQL standard. The keyword *unique* tells the system that the indexed field *SId* is a key of STUDENT. Thus the system knows that if we search SID_IDX for records having *SId*=10, then at most one record will be found, whereas if we search MAJOR_IDX for records having *MajorId*=10, then many records might be found.

```
create index MAJOR_IDX on STUDENT(MajorId);
create unique index SID_IDX on STUDENT(SId)
```

Figure 6-8: SQL statements to create indexes

Both of the indexes in Figure 6-8 are organized on a single field, which is most common. An index can also be organized on multiple fields, in which case all of the indexed fields are listed within the parentheses following the table name.

Conceptually, an index is a file of records. The file contains the same number of records as the table it is indexing. Each index record has a value for the indexed field(s), plus a special value called a *record identifier* (RID). An index record's *RID*-value points to its corresponding table record.

Figure 6-9 depicts our two example indexes and how they relate to the STUDENT table. For illustrative purposes, we show each index as sorted on its field.

SID_IDX STUDENT MAJOR_IDX

[redacted]

Figure 6-9: The SID_IDX and MAJOR_IDX indexes for STUDENT

Each SID_IDX record contains two fields: *SId* and *RID*. Similarly, each MAJOR_IDX record contains two fields: *MajorId* and *RID*. The *RID*-value of an index record points to the corresponding STUDENT record, and enables the system to access it without any additional search.

6.3.3 Using an index

An index is organized on its indexed fields. Thus if we are given values for these fields, then we can quickly find all index records having those values. And once we have an index record, we can use its *RID* value to find the corresponding table record. Note that this strategy does not require us to search the table at all, and does not depend on how the table happens to be organized.

The database system can use an index to implement a query that selects on the indexed field. For example, consider the SQL query of Figure 6-10(a), which retrieves the major of student #8. The database system can always implement this query by scanning the STUDENT table, as shown in Figure 6-10(b). However, the system can also use the SID_IDX index to implement the query, as shown in Figure 6-10(c).

```
select MajorId
from STUDENT
where SId = 8
```

(a) An SQL query to find the major of student #8

```
For each record in STUDENT:
   If the SId-value of this record is 8 then:
         Return the MajorId-value of that record.
```

(b) Implementing the query without an index

```
1. Use the SID_IDX index to find the index record
   whose SId-value is 8.
2. Obtain the value of its RID field.
3. Move directly to the STUDENT record having that RID.
4. Return the MajorId  value of that record.
```

(c) Implementing the query using an index

Figure 6-10: Implementing a query with and without an index

A cursory analysis suggests that the indexed implementation of this query is far more efficient. After all, the cost of searching the index is by definition small. (Recall that the binary search of Section 6.3.1 required us to examine at most 20 out of a million possible records.) And once we find the desired index record, we only need to look at one more record, namely the STUDENT record it points to. On the other hand, the scan of STUDENT may require that the system look at the entire table, which in a real database would be many thousands of records.

This cursory analysis gives us a feel for the value of the index, but it will be important for us to have a more exact estimate of the costs involved. So let's make some realistic assumptions about the size of STUDENT. We shall assume that:

- There are 45,000 records in STUDENT.
- There are 40 departments that a student can major in.
- Students are evenly divided amongst the majors. Thus there are 1,125 students per major.
- A disk block can contain 10 STUDENT records.

This last point requires some explanation. A *disk block* is the unit of read/write between the database system and the disk. So the database system will always read 10 STUDENT records at a time, whether it needs them or not. Furthermore, a disk operation is far more expensive than anything else that a database system does, which means that the best way to estimate the cost of a query is to count the number of disk blocks that are read.

With these assumptions, we can calculate the number of disk reads required by each implementation of the query in Figure 6-10.

- Scanning the STUDENT table requires 4,500 disk reads, in the worst case.
- Using the index requires a small number of disk reads to search the index, plus one disk read of the STUDENT table. In Section 21.4 we will see that a B-tree index on *SId* can be searched with only 2 disk reads, resulting in a total cost of just 3 disk reads[†].

In other words, using the index results in an implementation of this query that is 1,500 times faster than the unindexed implementation.

Let's try the same calculation on a different query. Figure 6-11(a) gives an SQL query that finds the IDs of the students having major #10. Figure 6-11(b) gives the implementation without an index, and Figure 6-11(c) gives the implementation using the MAJOR_IDX index.

```
select SId
from STUDENT
where MajorId = 10
```

(a) An SQL query to find the IDs of students having major #10

```
1. For each record in STUDENT:
      If the MajorId-value of this record is 10 then:
            Add the SId-value of that record to the output list.
2. Return the output list.
```

(b) Implementing the query without an index

```
1. Use the MAJOR_IDX index to find the index records whose
   MajorId-value is 10.
2. For each such index record:
      a.  Obtain the value of its RID field.
      b.  Move directly to the STUDENT record having that RID.
      c.  Add the SId-value of that record to the output list.
3. Return the output list.
```

(c) Implementing the query using an index

Figure 6-11: Implementing another query, with and without an index

[†] I don't know about you, but I find it remarkable that we only need 3 disk reads to find any record in a 4,500-block table. B-tree indexes are truly amazing. Skip ahead to Chapter 21 to find out how they work.

The cost of the non-indexed implementation is essentially the same as before. The only difference is that here, the system will always examine every STUDENT record. The cost of the implementation is 4,500 disk reads.

The cost of the indexed implementation is greater than in the previous example. The cost of searching the index is still small; let's assume again that it is only 2 disk reads. The cost of reading the STUDENT table, however, has increased significantly. Step 2b reads one STUDENT block from disk, and this step is executed for each of the 1,125 matching records. Thus the cost of the indexed implementation is a bit more than 1,125 disk reads, which is only about 4 times faster than the cost of the unindexed implementation.

Although both indexes are useful in these queries, it should be clear that SID_IDX is far more useful than MAJOR_IDX. The reason stems from the fact that the database system reads records from the disk in blocks. When the database system scans the STUDENT table directly, it gets to examine 10 records per disk read. This is a very efficient use of the disk. On the other hand, when the system reads the STUDENT table via the index, it examines just one of the records in that block, which is much less efficient. In general, the more matching records there are, the less efficiently the indexing strategy uses the disk.

To make this point even more clear, suppose that there are only 9 different departments, instead of 40. Then there will be 5,000 students in each major. If the database system uses the index to implement the query, it will need to read in one STUDENT block for each of the 5,000 matching students. However, recall that there are only 4,500 blocks in the STUDENT table. Consequently, the database system will actually be reading some blocks more than once! In this case, using the index is worse than simply scanning the STUDENT table directly. In fact, the index is completely useless.

We can sum up our observations in the following rule:

*The usefulness of an index on a field A is proportional to
the number of different A-values in the table.*

This rule implies that an index is most useful when its indexed field is the a key of the table (such as SID_IDX), because every record has a different key value. Conversely, the rule also implies that an index will be useless if the number of different *A*-values is less than the number of records per block. (See Exercise 6.16.)

6.3.4 Indexable queries

A given index can be used to implement many different queries. In this section we shall examine various kinds of indexable queries, using the index SID_IDX as an example.

The query of Figure 6-10(a) illustrated the first and most obvious kind of indexable query – namely, a query whose selection predicate equates the indexed field with a constant.

If the index is organized in sorted order, then it can also be used when the query's selection predicate compares the indexed field against a constant using "<" or ">". Such a query is called a *range query*. Figure 6-12 contains an example range query, together with an unindexed and indexed implementation of it.

```
select SName
from STUDENT
where SId > 8
```

(a) An SQL query to find the names of students having ID > 8

```
1. For each record in STUDENT:
      If the SId-value of this record is >8 then:
         Add the SName-value of that record to the output list.
2. Return the output list.
```

(b) Implementing the query without an index

```
1. Use the SID_IDX index to find the last index record whose
   SId-value is 8.
2. For the remaining records in the index (assuming that
   they are in sorted order):
      a.  Obtain the value of its RID field.
      b.  Move directly to the STUDENT record having that RID.
      c.  Add the SName-value of that record to the output list.
3. Return the output list.
```

(c) Implementing the query using an index

Figure 6-12: Implementing a range query

The usefulness of using an index to implement a range query depends on how many records satisfy the range predicate. In the case of Figure 6-12, if many students have an ID of more than 8, then it will be more effective to scan the STUDENT table directly.

An index can also be used to implement a join query. In this case, the join predicate must equate the indexed field with another field. Figure 6-13 gives an example join query and some possible implementations.

```
select s.SName, e.Grade
from STUDENT s, ENROLL e
where s.SId=e.StudentId
```

(a) An SQL query to find the names of students and their grades

```
1. For each record in ENROLL:
      For each record in STUDENT:
            If SId = StudentId then:
                  Add the values for SName and Grade to
                  the output list.
2. Return the output list.
```

(b) Implementing the query without an index

```
1. For each record in ENROLL:
      a.  Let x be the StudentId-value of that record.
      b.  Use the SID_IDX index to find the index record
          whose SId-value = x.
      c.  Obtain the value of its RID field.
      d.  Move directly to the STUDENT record having that RID.
      e.  Add the values for SName and Grade to the output list.
2. Return the output list.
```

(c) Implementing the query using an index

Figure 6-13: Implementing a join query

Both implementations of Figure 6-13 begin by scanning the ENROLL table. For each ENROLL record, the implementations need to find the matching STUDENT records. The non-indexed implementation does so by scanning the entire STUDENT table for each ENROLL record. The indexed implementation uses the *StudentId* value of each ENROLL record to search the index.

Let's compare the cost of these implementations. The non-indexed implementation must scan the entire 4,500-block STUDENT table for each ENROLL record. The indexed implementation requires a small number of disk reads (say, 3) for each ENROLL record. So the indexed implementation is about 1,500 times faster than the non-indexed one.

We can summarize the different uses of an index as follows:

An index on field A can be used to implement a query that has either:
- *a selection predicate that compares A with a constant; or*
- *a join predicate that equates A with another field.*

So far, the predicate in each of our example queries has been very simple, consisting of exactly what was needed to enable an index implementation. In general, however, a query predicate will also contain portions that have nothing to do with an index. In this case, the query processor can use the index to find the records satisfying part of the predicate, and then do a selection to check each of those records against the remainder of the predicate.

For example, consider the query of Figure 6-14(a). This query contains a selection predicate and a join predicate. The index SID_IDX is applicable to the selection predicate. So the implementation of Figure 6-14(b) uses the index first, to find student #8. It then scans the DEPT table, using the join predicate to find the matching department.

```
select d.DName
from STUDENT s, DEPT d
where s.MajorId=d.DId and s.SId=8
```

(a) An SQL query to find the name of student #8's major

```
1.  Use the SID_IDX index to find the index record
    having SId=8.
2.  Obtain the value of its RID field.
3.  Move directly to the STUDENT record having that RID.
    Let x be the MajorId-value of that record.
4.  For each record in DEPT:
      If the DId-value of that record = x then:
             Return the value of DName.
```

(b) Implementing the query using the index SID_IDX

```
1. For each record in DEPT:
     a.  Let y be the DId-value of that record.
     b.  Use the MAJOR_IDX index to find the index records
         whose MajorId-value = y.
     c.  For each such index record:
           • Obtain the value of its RID field.
           • Move directly to the STUDENT record having that RID.
           • If SId=8 then:
                Return the value of DName.
```

(c) Implementing the query using the index MAJOR_IDX

Figure 6-14: Two ways to implement a query using an index

An interesting thing about this query is that it also has an implementation that uses another index. This implementation appears in Figure 6-14(c). It uses the index

MAJOR_IDX to implement the join – For each DEPT record, the index retrieves the matching STUDENT records. The implementation then checks the *SId* value of each matching record; when the value 8 is located, then the *DName* of the current DEPT record is returned.

This example illustrates how a single query may be indexable in different ways. The query processor, as always, is responsible for determining which of these implementations is best. That process is called *query optimization*, and is the subject of Chapter 24.

6.3.5 Controlling redundancy

Indexes are entirely redundant. Each index record corresponds to a particular table record, and the values in that table record completely determine the values of the index record. The database system controls this redundancy by keeping the contents of each index synchronized with its corresponding table.

In particular, the database system must perform the following actions when the table is changed:

- If a record is inserted into a table, then a corresponding record must be inserted into each of its indexes.
- If a record is deleted from a table, then the corresponding record must be deleted from each of its indexes.
- If the field *A* of a record is modified, then only an index on *A* needs to be changed. The database system must modify the *A*-value of the corresponding record in that index.

This last point illustrates why an index is less expensive to maintain than a full copy of the table. A copy will always need to be modified when any field of a table record is modified, but an index will not need to change unless the modification is to an indexed field. For example, if the STUDENT table does not have an index on *GradYear* or *SName*, then modifying those values in the table will not require any corresponding modification to the indexes.

6.3.6 Deciding what indexes to create

Indexes are similar to materialized views, in that they should be invisible to users. The query processor makes the decisions about when to use an index. There is no way that a user can mention an index in a query, or force the query processor to use a particular index[†].

The database designer decides which indexes should be created. This decision should be based only on efficiency considerations. Each index has a benefit and a cost; an index should be created only if the benefit exceeds the cost.

[†] although some commercial systems allow a user to “suggest” that a particular index be used for a query.

We can compute the benefit of an index by multiplying two factors: the cost savings that the index provides for each query; and how frequently each query occurs. We have already seen how to calculate the cost savings of the index per query. That calculation requires knowing the number of records in the table, the number of records that fit in a disk block, and the number of distinct values of the indexed fields. Determining the frequency of indexable queries requires an analysis is similar to what the designer needs to do for materialized views – the designer determines the set of expected queries and their frequency, and then for each query determines what indexes would improve the implementation of that query.

The cost of keeping an index has two aspects.

One aspect is the disk space required to hold it. If the table has *N* fields of equal size, then each index will be slightly larger than *1/N* times the size of the table. So if we keep an index on every field, then the total size of the indexes will be somewhat larger than the size of the table. In many situations, disk space is cheap, and so the extra space usage is not significant. But if the database is very large, then it may not be practical to double the size of the database in order to keep an index on every field.

The other aspect is the time required to maintain it. If a data record is inserted or deleted, then the corresponding record in each index must likewise be inserted or deleted. So for example if a table contains three indexes, then a request to delete a record from the table will cause the system to delete four records – the table record plus the three index records. Similarly, a request to insert a new record into the table will cause the system to actually insert four records. In other words, the time required to process each update statement has been quadrupled.

This maintenance cost is related to the frequency of updates to the table. If the table is rarely updated (or if it is read-only), then maintenance is less of an issue, and more indexes are worth creating.

6.4 Chapter Summary

- Redundant data is valuable when it enables faster ways to execute queries.

- In order to avoid inconsistency problems, redundancy should be controlled by the database system. Two forms of controlled redundancy are *materialized views* and *indexes*.

- A materialized view is a table that contains the output of a view. The database system automatically updates the table whenever its underlying tables change. The update is performed *incrementally*.

- A view is worth materializing when its benefits outweigh its costs. The benefit of keeping a view increases when it is accessed frequently and its output is expensive to recompute each time. The cost increases when the view is updated frequently and its output is large.

- The database designer is responsible for determining which views to materialize. The designer must estimate the frequency of each query and each update. The designer specifies that a view is materialized by using a *create materialized view* statement.

- An old technique for improving query performance is called *denormalization*. Materialized views are a more modern solution to the same problem.

- The decision to use a materialized view in a query should be made by the query optimizer, not the user. Materialized views should be invisible to users.

- An *index* is associated with one or more fields of a table, and is used to improve the speed of queries that mention those fields.

- Each record in the index has a corresponding record in the table. An index record has a value for the indexed field(s), plus a special value called a *record identifier* (RID). The RID value in an index record points to its corresponding table record.

- An index is organized on its indexed fields, which allows the database system to quickly find all index records having specified values for those fields.

- Once the database system has found an index record, its can use the record's *RID* value to find the corresponding table record. Consequently, table records can be found quickly, without the need to search the table.

- An index is not necessarily useful. As a rule of thumb, the usefulness of an index on field *A* is proportional the number of different *A*-values in the table.

- An index on field *A* can be use to implement a query that has either a selection predicate that compares *A* with a constant, or a join predicate that equates *A* with another field.

- A query may be indexable in different ways. The query processor determines which of these implementations is best.

- The database system is responsible for updating the indexes when their table changes. For example, the system must insert (or delete) a record in each index whenever an record is inserted (or deleted) from the table. This maintenance cost means that only the most useful indexes are worth keeping.

6.5 Suggested Reading

The starting place for further investigation into materialized views should be [Gupta and Mumick 1999]. This book provides a detailed explanation of the principles, algorithms, and uses for materialized views. The article [Bello et al 1998] describes the implementation of materialized views in Oracle. And the article [Zhou et al. 2007] describes a technique for deferring updates to a materialized view until a query actually needs them.

The specification of materialized views and indexes are part of a larger topic called *physical database design*. A good coverage of the conceptual issues can be found in [Lightstone et al 2007]. *Database tuning* can be thought of as physical design after the database has been deployed. The book [Sasha and Bonnet 2002] covers the principles of tuning, including index specification. The tuning manual for the Derby database system can be found online at the URL *db.apache.org/derby/manuals*.

6.6 Exercises

CONCEPTUAL EXERCISES

6.1 Consider a materialized view whose definition joins a table with itself, as in:

```
define materialized view SAME_MAJOR_AS_STUDENT2 as
    select s1.SName
    from STUDENT s1, STUDENT s2
    where s1.MajorId=s2.MajorId and s1.SId=2
```

Explain how the database system can incrementally adjust the contents of this view when the STUDENT table changes.

6.2 Consider the materialized view of Figure 6-4, and suppose that it also calculated the sum and average of *GradYear*. Explain how the database system can incrementally adjust the contents of this view when the STUDENT table changes.

6.3 Give an algorithm for updating a materialized view defined by the following operators:
a) select
b) project
c) outer join
d) semijoin
e) antijoin
f) union

6.4 Consider the redundancy in the 3NF design of Figure 3.20(d). Show how to get a schema that is in BCNF, but has a materialized view for the FD enforcement.

6.5 Suppose that a designer wants to change the set of materialized views. For each of the following changes, explain what the impact will be on the users (if any).
a) The designer creates a new materialized view.
b) The designer changes a regular view to a materialized one.
c) The designer deletes an existing materialized view.
d) The designer changes an existing materialized view to a regular one.

6.6 Consider the assertions that were defined in Section 5.1. Show how a materialized view can be used to enforce an assertion efficiently.

6.7 Consider a database that has a fixed set of queries that are executed periodically, such as a monthly payroll report or a yearly budget of a corporation, or frequently, such as the current inventory of a store.
a) Explain the advantage of defining a materialized view for each such query.
b) Give circumstances under which it might not make sense to keep a materialized view on one or more of these queries.
c) Suppose that some of these queries shared a common subtask, such as performing the same joins or creating the same aggregation. Explain why it might make sense to create materialized views for those subtasks. How would you determine if it was a good idea?

6.8 Consider the following materialized view:

```
define materialized view MATH_STUDENT_INFO as
    select s.*, e.*
    from STUDENT s, ENROLL e
    where s.SId=e.StudentId and s.MajorId=10
```

a) For the following query, give an equivalent query that uses the materialized view.

```
select s.SName, k.Prof
from STUDENT s, ENROLL e, SECTION k
where s.SId=e.StudentId and e.SectionId=k.SectId
and s.MajorId=10 and e.Grade='F'
```

b) For the following query, give an equivalent query that uses the materialized view. (HINT: You need to use a UNION query.) Then explain why the materialized view is not useful in this case.

```
select s.SName
from STUDENT s
where s.MajorId=10
```

6.9 Consider the denormalized tables of Figure 6-5. Show that the tables are not key-based.

6.10 Consider the university database of Figure 1-1. Which fields would be inappropriate to index on? Explain your reasoning.

6.11 Explain which indexes could be useful for evaluating each of the following queries.
a)

```
select SName
from STUDENT, DEPT
where MajorId=DId and DName='math' and GradYear<>2001
```

b)

```
select Prof
from ENROLL, SECTION, COURSE
where SectId=SectionId and CourseId=CId
and Grade='F' and Title='calculus'
```

6.12 Calculate the cost of the queries of Figures 6-10 and 6-11 assuming the database contains only the records in Figure 6-9. Assume that 2 STUDENT records fit in a block.

6.13 Calculate the cost of the queries in Figure 6-11, assuming that there are only 5 different majors.

6.14 Suppose that we have decided to create an index on the field *GradYear* in STUDENT.
a) Give the appropriate SQL *create index* statement.
b) Consider the following query:

```
select * from STUDENT where GradYear=2004
```

Calculate the cost of this query, using the same assumptions as in Section 6.3.3, and assuming that students are evenly distributed among 50 different graduation years.
c) Do the same as in part (b), but instead of 50 different graduation years, use 2, 10, 20, and 100 different years.

6.15 Make reasonable assumptions for the other tables in the university database (or use the values in Figure 16-7). For each table, say which indexes are worth creating. Justify your decisions.

6.16 Section 6.3.3 asserted that an index on field *A* is useless if the number of different *A*-values is less than the number of table records in a block. Justify that assertion.

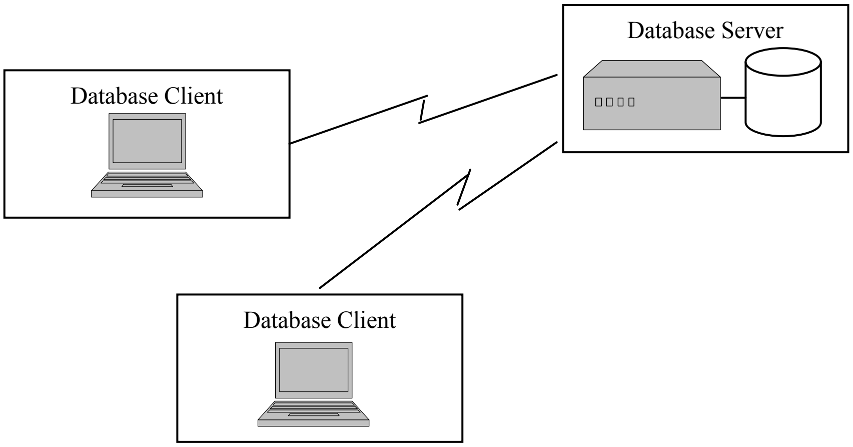
6.17 Figure 6-12(a) retrieved the majors of those students having *SId>8*. Suppose that we change the predicate to be "*SId<8*". Revise the implementation of Figure 6-12(c) accordingly.

6.18 Which of the implementations in Figure 6-14 do you think is faster? Explain why.

6.19 When a user creates a table, some database systems automatically create an index on the key of that table, as well as on every foreign key of that table.
a) One reason the database system creates these indexes is so that it can enforce the key and foreign key constraints. Explain how the indexes can help.
b) Give another reason why an index on the key ought to be worthwhile.
c) Give another reason why an index on a foreign key might be worthwhile.

PART 2
Client-Server Database Systems

Chapters 7-11 discuss the structure of a database system as seen by its users. In these chapters we examine the role of the database server, and how users write clients that interact with the server.



7
CLIENTS AND SERVERS

In this chapter we introduce the client-server database architecture, which is the most appropriate way to interact with a database when data needs to be shared among multiple users. We then examine the publicly-available database systems *Derby* and *SimpleDB*, both of which are client-server based. We show how to install and run each of the database servers, and how to run database clients against the servers.

7.1 The Data-Sharing Problem

Database systems arrived on the computer scene sometime in the mid 1960's. They lived on large mainframe computers, and like most software in those days, could be used only by specially-trained programmers. If someone wanted information from the database, that person would submit a request, and a programmer would (eventually) write a database program specifically for that request.

With the advent of personal computers came the personal database system. For a modest price, anybody can now own database software that not only is more powerful than the systems of the 1960's, but also is remarkably user-friendly. There is no need to write programs – users can interact with their database systems via nifty GUI interfaces.

Personal database systems are ideal for people who have a lot of personal data. But what happens if that data needs to be shared? One strategy is to physically distribute the actual database file. For example, governmental data (such as census and budget data), scientific data sets, and company product catalogs can all be found as files in various database formats. Once a user obtains the desired file, it can be loaded on the user's personal database system. However, this method of data sharing has two significant limitations: the data cannot be too large, and cannot be easily updated.

> *Physically distributing a personal database is not appropriate when the database is large or frequently updated.*

If the data is too large, then the user's machine will need significant disk space to hold it, as well as significant CPU speed and memory to execute queries efficiently. Each user will need to install the entire database, even if they are interested in only a small portion of it. To avoid these problems, a data provider can choose to split the database and create several smaller independent versions, with each version tailored to a specific use. But then a user having multiple needs will have to acquire multiple versions, and be responsible for any necessary re-integration of the data from different versions.

If shared data is copied onto each user's personal machine, then updates to one copy will not affect any other copy. The only way to incorporate changes into the database is for the data provider to create a new version of the database, and redistribute it. Three major problems arise:

- Users are unable to change the database directly. For example, if a user wishes to change personal information in the database, the user must send a request to the data provider, who will then (eventually) issue the update.
- The existence of multiple versions means that it is difficult (if not impossible) to ensure that all users have the most recent version of the data.
- It is usually impractical to prepare a new version for each update; typically, new updates are batched until the next version is created. Consequently, a user can never be sure that the data is accurate, even if the user has (or thinks he has) the most recent version of the database.

7.2 Database Clients and Servers

A better way to share data is to store it in a single location, on a machine that is accessible to all users. This machine is called the *server*. The server runs a program, called the *database server*, which is the only program that can access the database. A *database client* is a program that runs on a user's machine.

The data is shared by means of a *client-server protocol*. The database server program implements the server-side of the protocol, and a user's client program implements the client-side of the protocol. The server program is always running, waiting for connections from a client program. When a connection is established, the client sends the server a message requesting information. The server then accesses the database, and responds with an answer to the message. The client may send several requests to the server (and receive several responses) before closing the connection. Various protocols have been developed to support database access. When the client is a Java program, the protocol of choice is known as *JDBC*. Chapter 8 discusses JDBC in detail.

The server can be connected to multiple clients simultaneously. While the server is processing one client's request, other clients can be sending their own requests. The server contains a *scheduler*, which queues up requests waiting for service and determines when they get executed. Each client is unaware of the other clients, and (apart from delays due to scheduling) has the pleasant illusion that the server is dealing with it exclusively.

The client-server approach solves all of the above data-sharing problems:

- Because the data is centralized, there is no need for each user to download all of it; thus user machines do not require specialized resources. Only the server needs to be a high-end machine so that it can handle multiple simultaneous users and respond quickly to their requests.
- Users receive only the data they request, regardless of how large the database is. If, for efficiency, the server does choose to partition the database into smaller pieces, the partitioning is transparent to the user.

- Because there is only one copy of the data, that data can be updated immediately, giving users access to up-to-date information. Moreover, if the server allows clients to modify the database, then user updates can also be processed in real time.

> *A database server simplifies data sharing,*
> *because all activity is centralized in one place.*

The client-server approach also simplifies the construction of personal database systems, because it separates the user-interface components from the data-handling components. For example, Microsoft Access is constructed this way. It has a GUI front end (the client), and a back-end database engine (the server)[†]. The front end allows a user to interact with the database by clicking the mouse and filling in values. Whenever the front end determines that it needs information from the database, it constructs a request and sends it to the database engine, which then executes it and sends values back. Although the front and back ends in Access are closely coupled and run in the same process, their only interaction is via a formal client-server protocol. This system architecture allows the front end to be used with different back ends (e.g., an Access form can "link" to tables on another database server, which could be on a different machine), and the back end can be used with different front ends (e.g., an Excel spreadsheet can contain a macro that queries an Access database).

7.3 The Derby *and* SimpleDB *Database Servers*

This section examines how to install and run two different database servers: *Derby*, which is an open-source product of the Apache Foundation; and *SimpleDB*, which was written specifically for this text. The Derby server is full-featured, and implements the JDBC specifications and most of standard SQL. It is appropriate for executing SQL statements (as described in Part 1 of this book), and building database applications (as we shall see during Part 2). SimpleDB, on the other hand, has a much more limited functionality, and implements only a small fraction of standard SQL and JDBC. Its advantage is that its code is easy to read, understand, and modify. The SimpleDB code will be carefully scrutinized in Parts 3 and 4 of this book.

7.3.1 Installing and running Derby

The latest version of Derby can be downloaded from the Derby website, using the *downloads* tab at the URL *db.apache.org/derby*. The downloaded distribution file unpacks to a folder containing several directories – For example, the *docs* directory contains reference documentation, the *demo* directory contains a sample database, and so on. The full system contains many more features than we can cover here; the interested reader is encouraged to peruse the various guides and manuals in the *docs* directory.

In this book, we will be satisfied with a basic installation of Derby. This installation is quite simple; all you need to do is add four files to your Java classpath. These files are located in Derby's *lib* directory, and are named *derby.jar*, *derbynet.jar*, *derbyclient.jar*, and *derbytools.jar*.

[†] Microsoft calls this back-end component the "Jet database engine".

To install the Derby server, add the files derby.jar, derbynet.jar, derbyclient.jar, and derbytools.jar to your classpath.

Your classpath is determined from two sources: an environment variable called CLASSPATH, and an extension directory. Figure 7-1(a) and Figure 7-1(b) give directions for changing the CLASSPATH variable. Figure 7-1(c) explains how to use the extension directory.

In UNIX, the CLASSPATH variable is set in an initialization file, typically called *.bashrc*[†]. Assuming that the jar files are in the directory *~/derby/lib*, modify the file so that it has a line that looks like this:

```
CLASSPATH=.:~/derby/lib/derby.jar:~/derby/lib/derbynet.jar:~/derby/lib/
derbyclient.jar:~/derby/lib/derbytools.jar
```

Here, the ‘:’ character separates directory paths. (The string “.” is the first path, and denotes your current directory.) This classpath setting tells Java to search each of the 5 named directories when it needs to locate a class. (A jar file is treated like a directory.) If we don’t put the jar files in the classpath, then Java will not be aware that they exist and will never use them.

If your *.bashrc* file already contains a CLASSPATH setting, modify it to include the additional four paths.

(a) Directions for changing CLASSPATH in a UNIX system

In Windows, the CLASSPATH variable is set inside the *System* control panel. From that control panel, choose the *advanced* tab and click on the *environment variables* button. Assuming that the jar files are in the folder *C:\derby\lib*, create or modify a user variable named CLASSPATH whose value looks like this (and is all on one line):

```
.;C:\derby\lib\derby.jar;C:\derby\lib\derbynet.jar;C:\derby\lib\
derbyclient.jar;C:\derby\lib\derbytools.jar
```

Note that the ‘;’ character separates folder names in Windows environment variables.

(b) Directions for changing CLASSPATH in a Windows system

Each JVM has a specific directory, called its *extension directory*; each jar file in that directory is automatically added to the classpath of every program. So if you are the only user of your machine and you want to add a jar file to your classpath, then you can do so by simply placing it in this directory. You can determine the location of the directory by examining the value of the system property “java.ext.dirs”. Suppose for example that the home directory of your Java installation is in *C:\jdk*. Then your extension directory is probably located at *C:\jdk\jre\lib\ext*.

(c) Directions to change the system’s extension directory

Figure 7-1: Setting the CLASSPATH environment variable to install *Derby*

[†] This command works for LINUX using the *bash* shell. If your version of UNIX is different, a slightly different syntax may be necessary. Check with someone who understands the version of UNIX you are using.

The Derby server is written completely in Java, which means that we run it by executing its main class. This class is *NetworkServerControl*, in the package *org.apache.derby.drda*. Thus the following command starts the server:

```
   java org.apache.derby.drda.NetworkServerControl start -h localhost
```

This command specifies some arguments. The first argument is the string "start", which tells the class to start the server. We can stop the server by executing the same class with the argument string "shutdown".

The argument "-h localhost" tells the server to only accept requests from clients on the same machine. If you replace *localhost* by a domain name or IP address, then the server will only accept requests from that machine. Using the IP address "0.0.0.0" tells the server to accept requests from anywhere.[†]

Note that the command does not tell the server what database to use. Instead, that information is supplied by a client when it connects to the server. The Derby server is thus able to support multiple concurrent clients that access different databases.

The Derby databases are stored in a folder in the home directory of the user that started the server. When a client connects to the Derby server, the client specifies whether the specified database is an existing database or whether a new database should be created. If the client asks for an existing database where none exists, then the server will throw an exception.

If the server startup was successful, then the message "Apache Derby Network Server started and ready to accept connections" will be written to the console window.

7.3.2 Installing and running SimpleDB

The latest version of SimpleDB can be downloaded from its website at the URL *www.cs.bc.edu/~sciore/simpledb*. The downloaded file unpacks to a folder named *SimpleDB_x.y*; this folder contains directories *simpledb*, *javadoc*, and *studentClient*. The *simpledb* folder contains the server code. Unlike Derby, this code is not packed into a jar file; instead, every file is explicit within the folder.

To install the server, the path to the *SimpleDB_x.y* folder needs to be added to the classpath.

To install the SimpleDB server,
add the folder SimpleDB_x.y to your classpath.

[†] Of course, if you allow clients to connect from anywhere then you expose the database to hackers and other unscrupulous users. Typically, you would either place such a server inside of a firewall, enable Derby's authentication mechanism, or both.

Before you can run the SimpleDB server, you must first run the *rmiregistry* program (which is part of your Java SDK distribution) as a separate process. In UNIX, this means executing the command "`rmiregistry &`". In Windows, this means executing the command "`start rmiregistry`".

The SimpleDB server is written completely in Java; its main class is *Startup*, in the package *simpledb.server*. Thus the following command starts the server:

```
java simpledb.server.Startup studentdb
```

The argument to the command is the name of the database. Like Derby, the database is stored in a folder in the home directory of the user who starts the server. Unlike Derby, however, if a folder with the specified name does not exist, then a new folder will be created automatically for that database. Also unlike Derby, a SimpleDB server can handle only one database at a time; all clients must refer to that database.

The SimpleDB server accepts client connections from anywhere, corresponding to Derby's "*-h 0.0.0.0*" command-line option. The only way to shut down the server is to kill its process (for example, by closing its command window).

If the server startup was successful, then the message "database server ready" will be written to the console.

7.4 Running Database Clients

The SimpleDB distribution comes with several demo clients, in the *studentClient* folder. That folder contains two subfolders, corresponding to the Derby and SimpleDB servers. This section discusses how to run these clients against the database servers.

7.4.1 Running a client locally

Each client that ships with SimpleDB connects to its server using *localhost*; consequently, it can only be run locally from the server machine.

The client classes are not part of an explicit package, and thus they need to be run from the directory that they are stored in. For example, to run the *CreateStudentDB* client on the Derby server from Windows, you could execute the following commands:

```
> cd C:\SimpleDB_x.y\studentClient\derby
> java CreateStudentDB
```

The following list briefly describes the demo clients:

- *CreateStudentDB* creates and populates the student database shown in Figure 1-1. It must be the first client run on a new database.
- *StudentMajor* prints a table listing the names of students and their majors.
- *FindMajors* requires a command-line argument denoting the name of a department. The program then prints the name and graduation year of all students having that major.

- *SQLInterpreter* repeatedly prints a prompt asking you to enter a single line of text containing an SQL statement. The program then executes that statement. If the statement is a query, the output table is displayed. If the statement is an update command, then the number of affected records is printed. If the statement is ill-formed then an error message will be printed. The client runs equally well on either server. The only caveat is that the SimpleDB server understands a very limited subset of SQL, and will throw an exception if given an SQL statement that it does not understand. Section 7.6 describes the limitations of SimpleDB SQL.
- *ChangeMajor* changes Amy's STUDENT record to be a drama major. It is the only demo client that updates the database (although you can certainly use *SQLInterpreter* to run update commands).

The SimpleDB and Derby versions of each client are nearly identical. The only difference is that they use a different driver class, and with a different connection string.

The folder for the Derby server contains four additional subfolders:

- The *StudentInfo* folder contains an example of a Swing-based client. Executing the class *StudentInfo* brings up a Swing frame. If you enter the ID of a student in the textbox and click the SHOW INFO button, then the name and graduation year of the student is displayed, as well as information about the courses taken by that student. Clicking on the CHANGE GRADYEAR button opens a dialog box that asks you to enter a new graduation year; it then changes the student's record in the database, and displays that student's information. Clicking on the CLOSE button disconnects from the server and closes the application. This code is developed in Chapter 9.
- The *JPAStudentInfo* folder contains an alternative implementation of the *StudentInfo* client that uses the Java Persistence Architecture library. This code will be discussed in Chapter 9.
- The *XML* folder contains code to create an manipulate XML data. This code will be discussed in Chapter 10.
- The *Servlet* folder contains files that implement a webserver-based implementation of the *FindMajors* demo. These files cannot be executed directly; instead, they must be added to the configuration of a servlet-enabled web server. They are described in Chapter 11.

7.4.2 Running a client remotely

It is certainly convenient to run database clients on the same machine as the server, but also unrealistic and somewhat unsatisfying. To run a client program from a different machine, three things have to occur:

- You must configure the client machine properly.
- You must run the server program so that it will accept remote clients.
- You must modify the client program so that it contacts the desired server machine.

To configure a client machine, you must add the appropriate files to its classpath. But since the client does not run any of the server-side code, it only needs a subset of the files that the server requires. In particular: machines that run Derby clients only need to have the *derbyclient.jar* and *derbytools.jar* files in the classpath; and machines that run SimpleDB clients only need the package *simpledb.remote*.

To accept remote clients, the Derby server needs to be run with a "-*h*" command-line argument other than *localhost*. The SimpleDB server automatically accepts remote clients, so it requires no changes.

Finally, the client programs provided with SimpleDB need to be modified. Each program connects to the database server located at "localhost". In order to run one of the clients remotely, its source code must be modified so that the occurrence of *localhost* in the connection string is replaced by the domain name (or IP address) of the server machine.

7.5 The Derby ij *Client*

The *SQLInterpreter* demo client allows users to interactively execute SQL statements. Such a client is quite convenient, because it lets users access the database without having to write client programs themselves. Each server vendor typically provides its own SQL interpreter client for its users. Derby's client is called *ij*.

You run the *ij* client via the following command:

```
java org.apache.derby.tools.ij
```

The *ij* client reads and executes commands. A command is a string that ends with a semicolon. Commands can be split up over several lines of text; the *ij* client will not execute a command until it encounters a semicolon.

The first word of a command is the command name. SQL statements, which begin with "SELECT", "INSERT", "DELETE", etc., are all interpreted as commands. In addition, *ij* supports several other kinds of command.

Commands to connect, disconnect, and exit

Note that when the user runs the *ij* client, neither the name of the server machine or the name of the database is specified. Instead, *ij* commands allow the user to connect and disconnect from databases dynamically.

The argument to the *connect* command is a connection string, similar to what appears in a Derby client. For example, the following command connects to the existing *studentdb* database on the local server machine:

```
ij> connect 'jdbc:derby://localhost/studentdb;create=false';
```

Once you are connected to a server, you can then issue SQL statements. To disconnect from a server, you issue the *disconnect* command (with no arguments).

Similarly, you exit from the client by issuing the *exit* command.

Commands to manipulate cursors

The most interesting feature of *ij* is that it implements *cursor commands*. You can assign a cursor to an SQL query. The cursor will point to a single row of the query's output table. You can then use commands to move through the output table, examining desired rows and changing them if desired. We shall illustrate cursors via two examples.

For the first example, suppose we want to determine the two professors who have taught the most courses. Figure 7-2 displays a possible *ij* session.

```
ij> connect 'jdbc:derby://localhost/studentdb;create=false';
ij> get cursor busyprof as 'select Prof, count(SectId) as HowMany
                            from SECTION
                            group by Prof
                            order by HowMany desc';
ij> next busyprof;
PROF    |HOWMANY
--------------------
turing  |2

ij> next busyprof;
PROF    |HOWMANY
--------------------
newton  |1

ij> close busyprof;
ij> disconnect;
ij> exit;
```

Figure 7-2: An *ij* session to find the two professors who taught the most courses

In this figure, the *get* command assigns the cursor *busyprof* to the output of the specified query. Note that the query itself is in single quotes. The cursor is initially positioned before the first record of the output. The *next* command moves the cursor to the next output record, and displays it. Once those two records have been seen, the user closes the cursor.

The advantage to using cursors in this example is that it keeps the server from having to send too much data across the network to the client. For example, suppose we had issued the same query without using a cursor. Then the server would have sent the entire output table to the client, even though the user is only interested in the first two records.

For our second example, suppose that we want to go through the COURSE table, examining the courses and changing their titles as we see fit. Figure 7-3 displays a possible *ij* session, in which we decide to change the titles of the compilers and calculus courses.

```
ij> connect 'jdbc:derby://localhost/studentdb;create=false';
ij> get cursor crs as 'select * from COURSE for update';
ij> next crs;
CID    |TITLE        |DEPTID
--------------------------------
12     |db systems   |10

ij> next crs;
CID    |TITLE        |DEPTID
--------------------------------
22     |compilers   |10

ij> update COURSE set Title= 'interpreters'
    where current of crs;
1 row inserted/updated/deleted

ij> next crs;
CID    |TITLE        |DEPTID
--------------------------------
32     |calculus     |20

ij> update COURSE set Title= 'calc 1'
    where current of crs;
1 row inserted/updated/deleted

ij> disconnect;
ij> exit;
```

Figure 7-3: An *ij* session to change the titles of various courses

Note that the SQL query contains the clause "for update". This clause notifies the database server that the output records may get updated, and has two purposes. First, it lets the server verify that the query is updatable; a non-updatable query would cause *ij* to throw an exception. Second, it allows the server to obtain the necessary write-locks on the table.

The *where* clause of the update statement has the predicate "current of crs". This predicate is true only for the cursor's current record; it is thus an easy way to specify the record that we want changed.

7.6 The SimpleDB Version of SQL

The Derby server implements all of SQL. The SimpleDB server, on the other hand, implements only a tiny subset of standard SQL, and imposes restrictions not present in the SQL standard. In this section we briefly indicate what these restrictions are. Part 3 of the book goes into detail, and many end-of-chapter exercises will ask you to implement some of the omitted features.

A query in SimpleDB consists only of *select-from-where* clauses in which the *select* clause contains a list of fieldnames (without the AS keyword), and the *from* clause contains a list of tablenames (without range variables).

The terms in the optional *where* clause can be connected only by the Boolean operator *and*. Terms can only compare constants and fieldnames for equality. Unlike standard SQL, there are no other comparison operators, no other Boolean operators, no arithmetic operators or built-in functions, and no parentheses. Consequently, nested queries, aggregation, and computed values are not supported.

Because there are no range variables and no renaming, all field names in a query must be disjoint. And because there are no *group by* or *order by* clauses, grouping and sorting are not supported. Other restrictions are:

- The "*" abbreviation in the *select* clause is not supported.
- There are no null values.
- There are no explicit joins or outer joins in the *from* clause.
- The *union* keyword is not supported.
- *Insert* statements take explicit values only, not queries.
- *Update* statements can have only one assignment in the *set* clause.

7.7 Chapter Summary

- Personal database systems are inexpensive, powerful, and easy to use. However, they make it difficult to share data. A *client-server system* is ideal for data sharing, because all activity is centralized in one place.

- In a client-server system, the *database server* is a program that runs on a server machine; it is the only program that can access the database. A *database client* is a program that runs on a user's machine. The data is shared by means of a *client-server protocol*.

- Most database systems are implemented as a client-server system. Even personal database systems such as Microsoft Access can be run as servers.

- Two Java-based database servers are *Derby* and *SimpleDB*. Derby implements the full SQL standard, whereas SimpleDB implements only a limited subset of SQL. The benefit of SimpleDB is that its code is easy to understand, which will be useful for Parts 3 and 4 of this book.

- Derby is distributed with a client called *ij*, which allows users to interactively execute SQL statements. The client also lets users assign a *cursor* to a query. A cursor points to a single row of the query's output table, and can be used to iterate through the output table, one row at a time.

7.8 Suggested Reading

The client-server paradigm is useful in numerous areas of computing, not just databases. A general overview of the field can be found in [Orfali et al 1999]. Documentation on the various features and configuration options of the Derby server can be found at the URL *http://db.apache.org/derby/manuals/index.html.*

7.9 Exercises

CONCEPTUAL EXERCISES

7.1 Consider the database consisting of all telephone numbers and the people assigned to them. There are several ways to access this database: telephone directory books, computer CDs that contain all phone numbers for a particular area, web-based phone directory lookup pages (e.g. *whitepages.com*), and telephone-based information services (e.g. dialing "411" in the U.S.). Discuss the relative advantages of each way. Which are server-based? Which are more convenient? Which are more flexible? Which do you use, and why?

7.2 Server databases have a strong resemblance to the "old-fashioned" database systems of the mid 1960's, in that they both are centralized, run on powerful, high-end machines, and are accessed via programs.
a) In what ways did these old database systems satisfy the data-sharing needs of users? In what ways were they lacking?
b) What do these old database systems have that corresponds to the modern notion of a database client? In what ways are modern clients more effective?

PROGRAMMING EXERCISES

7.3 Install and run the Derby and SimpleDB servers.
a) Run their demo client programs locally from the server machine.
b) If you have access to a second machine, modify the demo clients and run them remotely from that machine as well.

7.4 The Derby system distributes its client support classes via the jar files *derbyclient.jar* and *derbytools.jar*. Actually, the file *derbytools.jar* is not needed in many cases. Use trial and error to figure out which activities of this chapter require that file.

7.5 The SimpleDB system distributes its client support classes in the *remote* folder. Create a jar file containing the *.class* files from that folder. Test your file by using it to install SimpleDB clients.

7.6 Create a UNIX shell script (or a Windows batch file) that will completely start up the SimpleDB server when run. (That is, it runs the *rmiregistry* program and the *simpledb.server.Startup* class.)

7.7 The class *simpledb.server.Startup* of the SimpleDB server requires that the name of the database be specified as a command-line argument. Revise the code so that the server uses a default database name if it is started without a command-line argument.

7.8 Use the Derby server and *ij* client to run the queries of Exercise 4.20 and 4.21 on the *studentdb* database.

7.9 User the Derby server and *ij* client to create a new videostore database, as described in Exercise 3.24. Then run the queries of Exercise 4.22. (HINT: The connection string for a new database ends with "*create=true*", and the connection string for an existing database ends with "*create=false*".)

8
USING JDBC
and the package java.sql

In Chapter 7 we learned that a client and server use a communication protocol to send requests and return results. Java clients interact with database servers via the protocol standard called *JDBC*.[†] As with SQL, JDBC contains many features that are useful for commercial applications but are unnecessary for learning about database systems. Consequently, we shall divide our study of JDBC into two parts: *basic JDBC*, which contains the classes and methods required for rudimentary usage; and *advanced JDBC*, which contains optional features that provide added convenience and flexibility.

8.1 Basic JDBC

The JDBC classes manage the transfer of data between a Java client and a database server. There are methods for connecting to the server, specifying the data to be transferred, managing the transfer, and translating between the SQL and Java representation of the data. Each of these tasks can have several ways to perform it, depending on how the database is being used. The JDBC API is therefore quite large.

The complete JDBC library consists of five packages, but we shall restrict our attention primarily to the package *java.sql*, which contains the core functionality. Moreover, the scope of the *java.sql* package is so enormous that we will need to focus on only a subset of it.

The principles of JDBC are embodied in just five interfaces: *Driver*, *Connection*, *Statement*, *ResultSet*, and *ResultSetMetadata*. Furthermore, only a very few methods of these interfaces are needed. We call this subset of JDBC *basic JDBC*. Figure 8-1 lists its API.

Basic JDBC consists of the interfaces and methods
that express the core of what JDBC does and how it is used.

[†] To many people, JDBC stands for *Java Data Base Connectivity*. However, Sun's legal position is that JDBC is not an acronym for anything.

Driver

```
public Connection connect(String url, Properties prop)
                                   throws SQLException;
```

Connection

```
public Statement createStatement() throws SQLException;
public void      close()           throws SQLException;
```

Statement

```
public ResultSet executeQuery(String qry) throws SQLException;
public int      executeUpdate(String cmd) throws SQLException;
```

ResultSet

```
public boolean   next()                throws SQLException;
public int       getInt()              throws SQLException;
public String    getString()           throws SQLException;
public void      close()               throws SQLException;
public ResultSetMetaData getMetaData() throws SQLException;
```

ResultSetMetaData

```
public int        getColumnCount()          throws SQLException;
public String     getColumnName(int column) throws SQLException;
public int        getColumnType(int column) throws SQLException;
public int getColumnDisplaySize(int column) throws SQLException;
```

Figure 8-1: The API for basic JDBC

The best way to understand how to use these interfaces is to look at some example JDBC programs. Our first example is the client *StudentMajor*, whose code appears in Figure 8-2. This client uses JDBC to execute a database query and print its output. The comments in the code demarcate the four activities that typically occur in this kind of program.

The four fundamental activities of a JDBC program are:

1. *Connect to the server.*
2. *Execute the desired query.*
3. *Loop through the resulting result set (for queries only).*
4. *Close the connection to the server.*

The following subsections discuss these four activities in detail.

```
import java.sql.*;
import org.apache.derby.jdbc.ClientDriver;

public class StudentMajor {
    public static void main(String[] args) {
      Connection conn = null;
      try {
         // Step 1: connect to database server
         Driver d = new ClientDriver();
         String url = "jdbc:derby://localhost/studentdb;create=false;";
         conn = d.connect(url, null);

         // Step 2: execute the query
         Statement stmt = conn.createStatement();
         String qry = "select SName, DName "
                    + "from DEPT, STUDENT "
                    + "where MajorId = DId";
         ResultSet rs = stmt.executeQuery(qry);

         // Step 3: loop through the result set
         System.out.println("Name\tMajor");
         while (rs.next()) {
            String sname = rs.getString("SName");
            String dname = rs.getString("DName");
            System.out.println(sname + "\t" + dname);
         }
         rs.close();
      }
      catch(SQLException e) {
         e.printStackTrace();
      }
      finally {
         // Step 4: Disconnect from the server
         try {
            if (conn != null)
               conn.close();
         }
         catch (SQLException e) {
            e.printStackTrace();
         }
      }
   }
}
```

Figure 8-2: The JDBC code for the *StudentMajor* client

8.1.1 Step 1: Connect to the database server

Each database server may have its own (and possibly proprietary) mechanism for making connections with clients. Clients, on the other hand, want to be as server-independent as possible. That is, a client doesn't want to know the nitty-gritty details of how to connect to a server; it simply wants the server to provide a class for the client to call. Such a class is called a *driver*.

In JDBC, a driver class implements the interface *Driver*. A client that connects to the Derby server uses the class *ClientDriver* (in package *org.apache.derby.jdbc*), whereas a SimpleDB client uses the class *SimpleDriver* (in package *simpledb.remote*).

A client connects to the database server on the specified host machine by calling the *connect* method of an appropriate *Driver* object. For example in Figure 8-2, this connection occurs via the following three lines of code:

```
Driver d = new ClientDriver();
String url = "jdbc:derby://localhost/studentdb;create=false;";
conn = d.connect(url, null);
```

The method *connect* takes two arguments. The first argument to the method is a URL that identifies the driver, the server, and the database. This URL is called the *connection string*. The connection string in the above code consists of four parts:

- The substring "jdbc:derby:" describes the protocol used by the driver. Here, the protocol says that this client is a Derby client that speaks JDBC.
- The substring "//localhost" describes the machine where the server is located. Instead of *localhost*, we could substitute any domain name or IP address.
- The substring "/studentdb;" describes the path to the database on the server. For a Derby server, the path begins at the home directory of the user that started the server.
- The remainder of the connection string consists of property values to be sent to the server. Here, the substring is "create=false;", which tells the server not to create a new database. In general, several property values can be sent to a Derby server. For example if the database server requires user authentication, then values for the properties *username* and *password* would also be specified.

The second argument to *connect* is an object of type *Properties*. This object provides another way to pass property values to the server. In the above code, the value of this argument is *null* because all properties are specified in the connection string. But we could have just as easily put the property specification into the second argument, as follows:

```
Driver d = new ClientDriver();
String url = "jdbc:derby://localhost/studentdb;";
Properties prop = new Properties();
prop.put("create", "false");
conn = d.connect(url, prop);
```

Each database server defines the syntax of its connection string. A SimpleDB connection string differs from Derby in that it contains only a protocol and machine name. (It doesn't make sense for the string to contain the name of the database, because the database is specified when the SimpleDB server is started. And the connection string doesn't specify properties because the SimpleDB server doesn't support any.) The following three lines of code illustrate how a client connects to a SimpleDB server:

```
        Driver d = new SimpleDriver();
        String url = "jdbc:simpledb://localhost";
        conn = d.connect(url, null);
```

Although the driver class and connection string are vendor-specific, the rest of a JDBC program is completely vendor-neutral. For example, consider the code of Figure 8-2, and examine the JDBC variables *d*, *conn*, *stmt*, and *rs*. We can tell that the variable *d* is assigned to a *ClientDriver* object. But what objects are assigned to the other variables? The variable *conn* is assigned to the *Connection* object that is returned by the method *connect*. However, since *Connection* is an interface, the actual class of this object is unknown to the client. Similarly, the client has no idea as to what objects are assigned to variables *stmt* and *rs*. Consequently, apart from the name of the driver class and its connection string, a JDBC program only knows about and cares about the vendor-neutral JDBC interfaces.

> *A basic JDBC client needs to import from two packages:*
> - *the built-in package java.sql, to obtain the vendor-neutral JDBC interface definitions;*
> - *the vendor-supplied package that contains the driver class.*

8.1.2 Step 2: Execute an SQL statement

A connection can be thought of as a "session" with the database server. During this session, the server executes SQL statements for the client. JDBC supports this mechanism as follows.

A *Connection* object has the method *createStatement*, which returns a *Statement* object. This *Statement* object contains two methods that execute SQL statements: *executeQuery* and *executeUpdate*.

The method *executeQuery* takes an SQL query as its argument. For example, the code of Figure 8-2 executes a query that joins STUDENT with DEPT. The method returns an object of type *ResultSet*. A result set represents the query's output table. The client can search through the result set to retrieve the desired information, as described in the next subsection.

The method *executeUpdate* executes an SQL update command as its argument, such as *create table*, *insert*, *delete*, and *update*. It executes the command and returns the number of records affected. For an example, the demo client *ChangeMajor* in Figure 8-3 executes an SQL statement to modify the *MajorId* value of a STUDENT record. Note that unlike queries, executing a command completes the command, because there is no output table to process.

```
public class ChangeMajor {
    public static void main(String[] args) {
      Connection conn = null;
      try {
         // step 1: connect to the database server
         Driver d = new ClientDriver();
         String url = "jdbc:derby://localhost/studentdb;create=false;";
         conn = d.connect(url, null);

         // Step 2: execute the update command
         Statement stmt = conn.createStatement();
         String cmd = "update STUDENT set MajorId=30 "
                    + "where SName = 'amy' ";
         stmt.executeUpdate(cmd);
         System.out.println("Amy is now a drama major.");

         // Step 3 is not needed for update commands
      }
      catch(SQLException e) {
         e.printStackTrace();
      }
      finally {
         // Step 4: disconnect from the server
         try {
            if (conn != null)
               conn.close();
         }
         catch (SQLException e) {
            e.printStackTrace();
         }
      }
   }
}
```

Figure 8-3: JDBC code for the *ChangeMajor* client

Step 2 of this code illustrates an interesting point. Since the SQL statement is stored as a Java string, it is encased in double quotes. SQL statements, on the other hand, use single quotes for their strings. This distinction makes our life easy, because we don't have to worry about a quote character having two different meanings – SQL strings use single quotes, and Java strings use double quotes.

8.1.3 Step 3: Loop through the output records

One a client obtains a result set, it iterates through the output records by calling the method *next*. This method moves to the next record, returning *true* if the move is successful and *false* if there are no more records. Typically, a client uses a loop to move through all of the records, processing each one in turn.

A ResultSet object can be thought of
as a collection of the records resulting from a query.

A new *ResultSet* object is always positioned before the first record, and so the typical loop structure looks like this:

```
String qry = "select ...";
ResultSet rs = stmt.executeQuery(qry);
while (rs.next()) {
      ... // process the record
}
rs.close();
```

An example of such a loop appears in the code of Figure 8-2. During the n^{th} pass through this loop, variable *rs* will be positioned at the n^{th} record of the result set. The loop will end when there are no more records to process.

When processing a record, a client uses the methods *getInt* and *getString* to retrieve the values of its fields. Each of the methods takes a fieldname as argument, and returns the value of that field. In Figure 8-2, the code retrieves and prints the values of fields *SName* and *DName* for each record.

A result set ties up valuable resources on the server, such as buffers and locks. The method *close* releases these resources and makes them available for other clients. A client should therefore strive to be a "good citizen" and always close its result sets as soon as possible.

8.1.4 Step 4: Disconnect from the server

The interface *Connection* also contains a method *close*. As with result sets, connections also hold valuable resources, and so a client should close a connection as soon as the database is no longer needed. Prompt closing is especially important in commercial database systems that restrict the number of simultaneous connections, because holding on to a connection could deprive another client from connecting.

The client program should close its connection even when an exception is thrown. A typical coding strategy, shown in Figures 8-2 and 8-3, is to close the connection within a *finally* block. This block will get executed at the end of the method, regardless of whether the method ends normally or not. One complication is that the *close* method itself can throw an exception, and so the *finally* block must contain an inner *try-catch* block. An exception thrown from the *finally* block would be truly serious – this could happen, for example, if the network were down. In such a case, there would be no way to close the connection. The approach taken in the demo code is to just give up and print the stack trace. Other possibilities are to wait and try again, or to send a message to a system administrator.

8.1.5 SQL Exceptions

The interaction between a client and database server can generate exceptions for many reasons. For example:

- The client cannot access the server. Perhaps the host name is wrong, or the host has become unreachable.
- The client asks the server to execute a badly-formed SQL statement.

- The client's query asks the server to access a non-existent table or to compare two incompatible values.
- The server aborts the client transaction because of deadlock.
- There is a bug in the server code.

Different database servers have their own internal way of dealing with these exceptions. SimpleDB, for example, throws a *RemoteException* on a network problem, a *BadSyntaxException* on an SQL statement problem, a *BufferAbortException* or *LockAbortException* on a deadlock, and a generic *RuntimeException* on a server problem.

In order to make exception handling vendor-independent, JDBC provides its own exception class, called *SQLException*. When a database server encounters an internal exception, it wraps it in an SQL exception and sends it to the client program.

The message string associated with an SQL exception specifies the internal exception that caused it. Each database server is free to provide its own messages. Derby, for example, has nearly 900 error messages, whereas SimpleDB lumps all of the possible problems into six messages: "network problem", "illegal SQL statement", "server error", "operation not supported", and two forms of "transaction abort".

SQL exceptions are checked, which means that clients must explicitly deal with them (either by catching them or throwing them onward). Consequently, a client will always get the opportunity to close its connection to the server.

8.1.6 Using query metadata

We define the *schema* of a result set to be the name, type, and display size of each field. This information is made available through the interface *ResultSetMetaData*.

When a client executes a query, it usually knows the schema of the output table. For example, hardcoded into the *StudentMajor* client is the knowledge that its result set contains the two string fields *SName* and *DName*.

However, suppose that a client allows users to submit queries as input. How then does that client know what fields to look for? It calls the method *getMetaData* on the result set. This method returns an object of type *ResultSetMetaData*. The client can then query this object about the output table's schema. The code in Figure 8-4 illustrates the use of the class to print the schema of an argument result set.

```
void printSchema(ResultSet rs) {
   ResultSetMetaData md = rs.getMetaData();
   for(int i=1; i<=md.getColumnCount(); i++) {
      String name  = md.getColumnName(i);
      int size     = md.getColumnDisplaySize(i);
      int typecode = md.getColumnType(i);
      String type;
      if (typecode == Types.INTEGER)
         type = "int";
      else if (typecode == Types.VARCHAR)
         type = "string";
      else
         type = "other";
      System.out.println(name + "\t" + type + "\t" + size);
   }
}
```

Figure 8-4: Using *ResultSetMetaData* to print the schema of a result set

This code uses its *ResultSetMetaData* object in a typical way. It first calls the method *getColumnCount* to return the number of fields in the result set; it then calls the methods *getColumnName*, *getColumnType*, and *getColumnDisplaySize* to determine the name, type, and size of a particular field. Note that column numbers start at 1, not 0 as you might expect.

The method *getColumnType* returns an integer that encodes the field type. These codes are defined as constants in the JDBC class *Types*. This class contains codes for 30 different types, which should give you an idea of how extensive the SQL language is. The actual values for these types are not important, because a JDBC program should always refer to the codes by name, not value.

A command interpreter is an example of a client that requires metadata knowledge. The demo program *SQLInterpreter* repeatedly reads SQL queries from the keyboard, executes them, and prints the output table; see Figure 8-5. As this is our first example of a non-trivial client, we will examine its code closely.

```
public class SQLInterpreter {
    private static Connection conn = null;

    public static void main(String[] args) {
      try {
         Driver d   = new ClientDriver();
         String url = "jdbc:derby://localhost/studentdb;create=false;";
         conn = d.connect(url, null);

         Reader rdr = new InputStreamReader(System.in);
         BufferedReader br = new BufferedReader(rdr);

         while (true) {
            // process one line of input
            System.out.print("\nSQL> ");
            String cmd = br.readLine().trim();
            System.out.println();
            if (cmd.startsWith("exit"))
               break;
            else if (cmd.startsWith("select"))
               doQuery(cmd);
            else
               doUpdate(cmd);
         }
      }
      catch (Exception e) {
         e.printStackTrace();
      }
      finally {
         try {
            if (conn != null)
               conn.close();
         }
         catch (Exception e) {
            e.printStackTrace();
         }
      }
   }

   private static void doQuery(String cmd) {
      try {
          Statement stmt = conn.createStatement();
          ResultSet rs = stmt.executeQuery(cmd);
          ResultSetMetaData md = rs.getMetaData();
          int numcols = md.getColumnCount();
          int totalwidth = 0;

          // print header
          for(int i=1; i<=numcols; i++) {
            int width = md.getColumnDisplaySize(i);
            totalwidth += width;
            String fmt = "%" + width + "s";
            System.out.format(fmt, md.getColumnName(i));
          }
          System.out.println();
          for(int i=0; i<totalwidth; i++)
             System.out.print("-");
          System.out.println();

          // print records
          while(rs.next()) {
            for (int i=1; i<=numcols; i++) {
               String fldname = md.getColumnName(i);
               int fldtype = md.getColumnType(i);
               String fmt = "%" + md.getColumnDisplaySize(i);
               if (fldtype == Types.INTEGER)
                  System.out.format(fmt + "d", rs.getInt(fldname));
               else
                  System.out.format(fmt + "s", rs.getString(fldname));
            }
            System.out.println();
         }
         rs.close();
      }
      catch (SQLException e) {
         System.out.println("SQL Exception: " + e.getMessage());
         e.printStackTrace();
      }
   }

   private static void doUpdate(String cmd) {
      try {
          Statement stmt = conn.createStatement();
          int howmany = stmt.executeUpdate(cmd);
          System.out.println(howmany + " records processed");
      }
      catch (SQLException e) {
         System.out.println("SQL Exception: " + e.getMessage());
         e.printStackTrace();
      }
   }
}
```

Figure 8-5: The JDBC code for the *SQLInterpreter* client

The *main* method processes one line of text during each iteration of its while loop. If the text is an SQL statement, the private method *doQuery* or *doUpdate* is called, as appropriate. The user can exit the loop by entering “exit”, at which point the program exits.

The method *doQuery* executes the query and obtains the result set and metadata of the output table. Most of the method is concerned with determining proper spacing for the values. The calls to *getColumnDisplaySize* return the space requirements for each field; the code uses these numbers to construct a format string that will allow the field values to line up properly. The complexity of this code illustrates the maxim “the devil is in the details”. That is, the conceptually difficult tasks are easily coded, thanks to the methods of *ResultSet* and *ResultSetMetaData*, whereas the trivial task of lining up the data takes most of the coding effort.

The methods *doQuery* and *doUpdate* trap exceptions by printing an error message and returning. This error-handling strategy allows the main loop to continue to accept statements until the user enters the “exit” command.

8.2 Advanced JDBC

Basic JDBC is relatively simple to use, but it provides a fairly limited set of ways to interact with the database server. In this section we consider some additional features of JDBC that give the client more control over how the database is accessed.

8.2.1 Hiding the driver

In basic JDBC, a client connects to a database server by obtaining an instance of a *Driver* object and calling its *connect* method. One problem with this strategy is that it places vendor-specific code into the client program. There are two vendor-neutral JDBC classes for keeping the driver information out of the client programs: *DriverManager* and *DataSource*. We shall discuss each in turn.

Using *DriverManager*

The class *DriverManager* holds a collection of drivers. It contains methods to add a driver to the collection, and to search the collection for a driver that can handle a given connection string. The API for these methods appears in Figure 8-6.

```
static public void registerDriver(Driver driver)
                                         throws SQLException;

static public Connection getConnection(String url, Properties p)
                                         throws SQLException;

static public Connection getConnection(String url)
                                         throws SQLException;

static public Connection getConnection(String url,
                                       String user, String pwd)
                                         throws SQLException;
```

Figure 8-6: Part of the API for the class *DriverManager*

The idea is that the client repeatedly calls *registerDriver* to register the driver for each database that it might use. When the client wants to connect to a database, it only needs to call the *getConnection* method and provide it with a connection string. The driver manager tries the connection string on each driver in its collection until one of them returns a non-null connection.

The *DriverManager* class provides three versions of the method *getConnection*. The first version, whose second argument is a property list, corresponds to the arguments of the method *connect* in *Driver*. The other versions are convenience methods. The second version can be used when there are no properties, and the third can be used when the only properties are the user name and password.

For example, consider the code of Figure 8-7. The first two lines register the Derby and SimpleDB drivers with the driver manager. The last two lines establish a connection to the Derby server. The client does not need to specify the driver when it calls *getConnection*; it only specifies the connection string. The driver manager determines, from its registered drivers, which one to use.

```
DriverManager.registerDriver(new ClientDriver());
DriverManager.registerDriver(new SimpleDriver());

String url   = "jdbc:derby://localhost/studentdb;create=false;";
Connection c = DriverManager.getConnection(url);
```

Figure 8-7: Connecting to a Derby server using *DriverManager*

The use of *DriverManager* in Figure 8-7 is not especially satisfying, because the driver information hasn't been hidden – it is right there in the calls to *registerDriver*. JDBC resolves this issue by allowing the drivers to be specified in the Java system-properties file. For example, the Derby and SimpleDB drivers can be registered by adding the following line to the file:

```
jdbc.drivers=org.apache.derby.jdbc.ClientDriver:simpledb.remote.SimpleDriver
```

Placing the driver information in the properties file is an elegant solution, because it takes the driver specification out of the client code and puts it in a central location. Consequently by changing this one file, we can revise the driver information used by all JDBC clients, and without having to recompile any code.

Using *DataSource*

Although the driver manager can hide the drivers from the JDBC clients, it cannot hide the connection string. A more recent addition to JDBC is the interface *DataSource*, which is in the package *javax.sql*. This is currently the preferred strategy for managing drivers.

A *DataSource* object encapsulates both the driver and the connection string. A client can therefore connect to a server without knowing any of the connection details. For example, the client code might look like this:

```
   DataSource ds = makeDerbySource("localhost", "studentdb", null);
   Connection conn = ds.getConnection();
```

This code calls a local method *makeDerbySource* to create its data source. The code for this method appears in Figure 8-8. This method takes three arguments, denoting the location of the server, the name of the database, and the property list. It returns a *DataSource* object that will connect to the specified database on a Derby server.

```
DataSource makeDerbySource(String host, String db, Properties p) {
   ClientDataSource cds = new ClientDataSource();
   cds.setServerName(host);
   cds.setDatabaseName(db);
   cds.setConnectionAttributes(p);
   return cds;
}
```

Figure 8-8: Creating a data source for a Derby server

Figure 8-8 depends on the class *ClientDataSource*. This class is supplied by Derby, and implements the *DataSource* interface. Each database vendor supplies its own class that implements *DataSource*. Since this class is vendor-specific, it knows the details of its driver, such as the driver name and the syntax of the connection string. Therefore the code that uses this class only needs to specify the requisite properties. For example, the *ClientDataSource* class has methods *setServerName* and *setDatabaseName* for setting specific connection properties. It also has a method *setConnectionAttributes* for setting ancillary properties, such as whether a new database should be created. Note that the new *ClientDataSource* object is returned as a *DataSource*. As a result, the client using the data source is not aware of its actual class or the values of its properties.

The nice thing about using *ClientDataSource* to create a data source is that the client no longer needs to know the name of the driver and the syntax of the connection string. On the other hand, the class is still vendor-specific, and thus the client code is not completely vendor-independent. This problem can be addressed in a couple of ways.

One solution is for the database administrator to save the *DataSource* object in a file. The DBA can create the object, and use Java serialization to write it to the file. A client can then obtain the data source by reading the file and de-serializing it back to a *DataSource* object. This solution is similar to using a properties file. Once the *DataSource* object is saved in the file, it can be used by any JDBC client. And the DBA can make changes to the data source by simply replacing the contents of that file.

A second solution is to use a name server (such as a JNDI server) instead of a file. The DBA places the *DataSource* object on the name server, and clients then request the data source from the server. Given that name servers are a common part of many computing environments, this solution is often easy to implement. This solution is especially useful in a web server environment, as we shall see in Chapter 11.

8.2.2 Explicit transaction handling

Each JDBC client runs as a series of *transactions*. Conceptually, a transaction is a "unit of work," meaning that all of its database interactions are treated as a unit. For example, if one update in a transaction fails, then the server will ensure that all updates made by that transaction will fail.

A transaction *commits* when its current unit of work has completed successfully. The database server implements a commit by making all modifications permanent, and releasing any resources (e.g. locks) that were assigned to that transaction. Once the commit is complete, the server starts a new transaction.

A transaction *rolls back* when it cannot commit. The database server implements a rollback by undoing all changes made by that transaction, releasing locks, and starting a new transaction.

A transaction has not completed
until it has either committed or rolled back.

Although transactions are an essential part of JDBC, they are not explicit in basic JDBC. Instead, the database server chooses the boundaries of each transaction, and decides whether a transaction should be committed or rolled back. This situation is called *autocommit*.

During autocommit, the server executes each SQL statement in its own transaction. The server commits the transaction if the statement successfully completes, and rolls back the transaction otherwise. Note that an update command completes as soon as the *executeUpdate* method has finished, but a query will still be active after the *executeQuery* method finishes. In fact, a query must stay active until the client has finished looking at the output records. Consequently, a query doesn't complete until the client closes the query's result set.

A transaction accrues locks, which are not released until the transaction has committed or rolled back. Because these locks can cause other transactions to wait, a client should strive for short transactions. This principle implies that a client running in autocommit mode should always close its result sets as soon as possible.

Autocommit is a reasonable default mode for JDBC clients. Having one transaction per statement leads to short transactions, and often is the right thing to do. However, there are circumstances when a transaction ought to consist of several SQL statements.

One situation where autocommit is undesirable is when a client needs to have two statements active at the same time. For example, consider the code fragment of Figure 8-9. This code first executes a query that retrieves all courses. It then loops through each output record, asking the user whether the course should be deleted. If so, it executes an SQL deletion statement to do so.

```
DataSource ds = ...
Connection conn = ds.getConnection();
Statement stmt1 = conn.createStatement();
Statement stmt2 = conn.createStatement();
ResultSet rs = stmt1.executeQuery("select * from COURSE");
while (rs.next()) {
   String title = rs.getString("Title");
   boolean goodCourse = getUserDecision(title);
   if (!goodCourse) {
      int id = rs.getInt("CId");
      stmt2.executeUpdate("delete from COURSE where CId =" + id);
   }
}
rs.close();
```

Figure 8-9: Code that may behave incorrectly in autocommit mode

The problem with this code is that the deletion statement will be executed while the record set is still open. Because each SQL statement runs in its own transaction, the server must preemptively commit the query's transaction before it can create a new transaction to execute the deletion. And since the query's transaction has committed, it doesn't really make sense to access the remainder of the record set. The code will either throw an exception or have unpredictable behavior[†].

Another situation where autocommit is undesirable occurs when multiple modifications to the database need to happen together. The code fragment of Figure 8-10 provides an example. The intent of the code is to swap the professors teaching sections 43 and 53. However, if the server crashes after the first call to *executeUpdate* but before the second one, then the database will be incorrect. Here it is important that both SQL statements occur in the same transaction, so that they are either committed together or rolled back together.

[†] The actual behavior of this code depends on the *holdability* of the result set, whose default value is server dependent. If the holdability is CLOSE_CURSORS_AT_COMMIT, then the result set will become invalid, and an exception will be thrown. If the holdability is HOLD_CURSORS_OVER_COMMIT, then the result set will stay open, but its locks will be released. The behavior of such a result set is unpredictable, and similar to the read-uncommitted isolation mode that we will see in Section 7.2.3.

```
DataSource ds = ...
Connection conn = ds.getConnection();
Statement stmt = conn.createStatement();
String cmd1 = "update SECTION set Prof= 'brando' where SectId = 43";
String cmd2 = "update SECTION set Prof= 'einstein' "
            + "where SectId = 53";

stmt.executeUpdate(cmd1);
// suppose that the server crashes at this point
stmt.executeUpdate(cmd2);
```

Figure 8-10: Code that could behave incorrectly in autocommit mode

Autocommit mode can also be inconvenient. Consider a client that is performing multiple insertions, say by loading data from a text file. If the server crashes while the client is running, then some of the records will be inserted and some will not. It could be tedious and time-consuming to determine where the client failed, and to rewrite the client to insert only the missing records. However, if we instead placed all of the insertion commands in the same transaction, then all of them would get rolled back, and it would be possible to simply rerun the client.

The *Connection* interface contains three methods that allow the client to get out of autocommit mode and handle its transactions explicitly. Figure 8-11 gives their API.

```
public void setAutoCommit(boolean ac) throws SQLException;
public void commit()                  throws SQLException;
public void rollback()                throws SQLException;
```

Figure 8-11: The *Connection* API for handling transactions explicitly

A client turns off autocommit by calling *setAutoCommit(false)*. This method call causes the server to start a new transaction. The client can complete the current transaction and start a new one by calling *commit* or *rollback*, as desired.

When a client turns off autocommit, it takes on the responsibility for rolling back failed SQL statements. In particular, if an exception gets thrown during a transaction, then the client must roll back that transaction inside of its exception-handling code.

For an example, consider again the incorrect code fragment of Figure 8-9. A corrected version appears in Figure 8-12.

```
Connection conn = null;
try {
   DataSource ds = ...
   Connection conn = ds.getConnection();
   conn.setAutoCommit(false);

   Statement stmt = conn.createStatement();
   ResultSet rs = stmt.executeQuery("select * from COURSE");
   while (rs.next()) {
      String title = rs.getString("Title");
      boolean goodCourse = getUserDecision(title);
      if (!goodCourse) {
         int id = rs.getInt("CId");
         stmt.executeUpdate("delete from COURSE where CId =" + id);
      }
   }
   rs.close();
   conn.commit();
}
catch (SQLException e) {
   try {
      e.printStackTrace();
      conn.rollback();
   }
   catch (SQLException e2) {
      System.out.println("Could not roll back");
   }
}
finally {
   try {
      if (conn != null)
         conn.close();
   }
   catch (Exception e) {
      e.printStackTrace();
   }
}
```

Figure 8-12: A revision of Figure 8-9 that handles transactions explicitly

Note that the code calls *setAutoCommit* immediately after the connection is created, and calls *commit* immediately after the statements have completed. The *catch* block contains the call to *rollback*. This call needs to be placed inside its own *try* block, in case it throws an exception. Of course, an exception during rollback is a potentially serious issue, because it means that the database could have become corrupted (as we saw in Figure 8-11). This code simply prints an error message, although in general more drastic measures might be appropriate.

8.2.3 Setting transaction isolation level

A database server typically has several clients active at the same time, each with their own transaction. By executing these transactions concurrently, the server can improve their throughput and response time. Thus, concurrency is a good thing. However, *uncontrolled* concurrency can cause problems, because a transaction can interfere with another transaction by modifying the data it uses in unexpected ways.

We will present a series of three examples that demonstrate the kinds of problems that can occur.

Example 1: Reading Uncommitted Data

Consider again the code for Figure 8-10, which swaps the professors of two sections, and assume that it runs as a single transaction (i.e., with autocommit turned off). Call this transaction T_1. Suppose also that the university has decided to give bonuses to its professors, based on the number of sections taught; it therefore executes a transaction T_2 that counts the sections taught by each professor. Furthermore, suppose that these two transactions happen to run concurrently – in particular, suppose that T_2 begins and executes to completion immediately after the first update statement of T_1. The result is that Professors Brando and Einstein will get credited respectively with one extra and one fewer course than they deserve, which will affect their bonuses.

What went wrong? Each of the transactions is correct in isolation, but together they cause the university to give out the wrong bonuses. The problem is that T_2 incorrectly assumed that the records it read were *consistent* – that is, that they made sense together. However, data written by an uncommitted transaction may not always be consistent. In the case of T_1, the inconsistency occurred at the point where only one of the two modifications was made. When T_2 read the uncommitted modified record at that point, the inconsistency caused it to make incorrect calculations.

Example 2: Unexpected Changes to an Existing Record

For this example, assume that the STUDENT table contains a field *MealPlanBalance*, which denotes how much money the student has for buying food in the cafeteria. Consider the two transactions of Figure 8-13. Transaction T_1 executed when Joe bought a $10 lunch. The transaction runs a query to find out his current balance, verifies that the balance is sufficient, and decrements his balance appropriately. Transaction T_2 executed when Joe's parents sent in a check for $1,000, to be added to his meal plan balance. That transaction simply runs an SQL update statement to increment Joe's balance.

```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select MealPlanBalance "
                               + "from STUDENT where SId = 1");
rs.next();
int balance = rs.getInt("MealPlanBalance");
rs.close();

int newbalance = balance - 10;
if (newbalance < 0)
   throw new NoFoodAllowedException("You cannot afford this meal");

stmt.executeUpdate("update STUDENT "
                 + "set MealPlanBalance = " + newbalance
                 + " where SId = 1");
conn.commit();
```

(a) Transaction T_1 decrements the meal plan balance

```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
stmt.executeUpdate("update STUDENT "
                 + "set MealPlanBalance = MealPlanBalance + 1000 "
                 + "where SId = 1");
conn.commit();
```

(b) Transaction T_2 increments the meal plan balance

Figure 8-13: Two concurrent transactions that can manage to "lose" an update

Now suppose that these two transactions happen to run concurrently at a time when Joe has a $50 balance. Moreover, suppose that T_2 begins and executes to completion immediately after T_1 calls *rs.close*. Then T_2, which commits first, will modify the balance to $1050. However, T_1 is unaware of this change, and still thinks that the balance is $50. It thus modifies the balance to $40 and commits. The result is that the $1,000 deposit is not credited to his balance; that is, the update got "lost".

The problem here is that transaction T_1 incorrectly assumed that the value of the meal plan balance would not change between the time that T_1 read the value and the time that T_1 modified the value. Formally, this assumption is called *repeatable read*, because the transaction assumes that repeatedly reading an item from the database will always return the same value.

<u>Example 3: Unexpected Changes to the Number of Records</u>

Suppose that the university dining services made a profit of $100,000 last year. The university feels bad that it overcharged its students, so it decides to divide the profit equally among them. That is, if there are 1,000 current students, then the university will add $100 to each meal plan balance. The code appears in Figure 8-14.

```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
String qry = "select count(SId) as HowMany "
           + "from STUDENT "
           + "where GradYear >= extract(year, current_date)"
ResultSet rs = stmt.executeQuery(qry);
rs.next();
int count = rs.getInt("HowMany");
rs.close();

int rebate = 100000 / count;
String cmd = "update STUDENT "
           + "set MealPlanBalance = MealPlanBalance + " + rebate
           + " where GradYear >= extract(year, current_date)";
stmt.executeUpdate(cmd);
conn.commit();
```

Figure 8-14: A transaction that could give out more rebates than expected

The problem with this transaction is that it assumes that the number of current students will not change between the calculation of the rebate amount and the updating of the STUDENT records. But suppose that several new STUDENT records got inserted into the database between the closing of the recordset and the execution of the update statement. Not only will these new records incorrectly get the precalculated rebate, but the university will wind up spending more than $100,000 on rebates. These new records are known as *phantom records*, because they mysteriously appear after the transaction has started.

These examples illustrate the kind of problems that can arise when two transactions interact. The only way to guarantee that an arbitrary transaction will not have problems is to execute it in complete isolation from the other transactions. This form of isolation is called *serializability*, and is discussed in considerable detail in Chapter 14.

Unfortunately for us, serializable transactions can run very slowly, because they require the database server to significantly reduce the amount of concurrency it allows. In fact, they can run so slowly that JDBC does not force transactions to use serializable isolation; instead it lets clients specify how much isolation they want their transaction to have.

JDBC defines four isolation levels:

- *Read-Uncommitted* isolation means no isolation at all. Such a transaction could suffer any of the problems from the above three examples.

- *Read-Committed* isolation forbids a transaction from accessing uncommitted values. Problems related to nonrepeatable reads and phantoms are still possible.
- *Repeatable-Read* isolation extends read-committed so that reads are always repeatable. The only possible problems are due to phantoms.
- *Serializable* isolation guarantees that no problems will ever occur.

A JDBC client specifies the isolation level it wants by calling the method *Connection.setTransactionIsolation*. For example, the following code fragment sets the isolation level to serializable:

```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

These four isolation levels exhibit a tradeoff between execution speed and potential problems. That is, the faster we want our transaction to run, the greater the risk we must accept that the transaction might run incorrectly. This risk can be mitigated by a careful analysis of the client.

For example, you might be able to convince yourself that phantoms or nonrepeatable reads cannot be a problem. This would be the case, for example, if your transaction performs only insertions, or if it deletes specific existing records (as in "delete from STUDENT where SId=1"). In this case, an isolation level of read-committed will be fast and correct.

For another example, you might convince yourself that any potential problems are uninteresting. Suppose that your transaction calculates, for each year, the average grade given during that year. You decide that even though grade changes can occur during the execution of the transaction, those changes are not likely to affect the resulting statistics significantly. In this case, you could reasonably choose the isolation level of read-committed, or even read-uncommitted.

The default isolation level for many database servers (including Derby, Oracle, and Sybase) is read-committed. This level is appropriate for the simple queries posed by naïve users in autocommit mode. However, if your client programs perform critical tasks, then it is equally critical that you carefully determine the most appropriate isolation level.

A programmer that turns off autocommit mode must be very careful to choose the proper isolation level of each transaction.

8.2.4 Prepared statements

Many JDBC client programs are *parameterized*, in the sense that they accept an argument value from the user, and execute an SQL statement based on that argument. An example of such a client is the demo client *FindMajors*, whose code appears in Figure 8-15.

```
public class FindMajors {
    public static void main(String[] args) {
      String major = args[0];
      System.out.println("Here are the " + major + " majors");
      System.out.println("Name\tGradYear");

      Connection conn = null;
      try {
         // Step 1: connect to database server
         Driver d = new ClientDriver();
         String url = "jdbc:derby://localhost/studentdb;create=false;";
         conn = d.connect(url, null);

         // Step 2: execute the query
         Statement stmt = conn.createStatement();
         String qry = "select SName, GradYear "
                    + "from STUDENT, DEPT "
                    + "where DId = MajorId "
                    + "and DName = '" + major + "'";
         ResultSet rs = stmt.executeQuery(qry);

         // Step 3: loop through the result set
         while (rs.next()) {
            String sname = rs.getString("SName");
            int gradyear = rs.getInt("GradYear");
            System.out.println(sname + "\t" + gradyear);
         }
         rs.close();
      }
      catch(Exception e) {
         e.printStackTrace();
      }
      finally {
         // Step 4: close the connection to the server
         try {
            if (conn != null)
               conn.close();
         }
         catch (SQLException e) {
            e.printStackTrace();
         }
      }
   }
}
```

Figure 8-15: The JDBC code for the *FindMajors* client

A user runs this client by supplying a department name on the command line. The code then incorporates this name into the SQL query that it executes. For example, suppose that the client was called with the command-line argument *math*. Then the SQL query generated in step 2 would be:

```
select SName, GradYear
from STUDENT, DEPT
where DId = MajorId and DName = 'math'
```

Note how the client code in Figure 8-15 explicitly adds the single quotes surrounding the department name.

Instead of generating an SQL statement dynamically this way, our client can use a *parameterized SQL statement*. A parameterized statement is an SQL statement in which '?' characters denote missing parameter values. A statement can have several parameters, all denoted by '?'. Each parameter has an *index value* that corresponds to its position in the string. For example, the following parameterized statement deletes all students having a yet-unspecified graduation year and major:

```
delete from STUDENT where GradYear = ? and MajorId = ?
```

In this statement, the value for *GradYear* is assigned index 1, and the value for *MajorId* is assigned index 2.

The JDBC class *PreparedStatement* handles parameterized statements. A client processes a prepared statement in three steps:

- It creates a *PreparedStatement* object for a specified parameterized SQL statement.
- It assigns values to the parameters.
- It executes the prepared statement.

For example, Figure 8-16 gives step 2 of the *FindMajors* client, revised to use prepared statements.

```
// Step 2: execute the query
String qry = "select SName, GradYear "
           + "from STUDENT, DEPT "
           + "where DId = MajorId and DName = ?";
PreparedStatement pstmt = conn.prepareStatement(qry);
pstmt.setString(1, major);
ResultSet rs = pstmt.executeQuery();
```

Figure 8-16: Revising the *FindMajors* client to use prepared statements

The last three statements of this code correspond to the above three bullet points. First, the client creates the *PreparedStatement* object, by calling the method *prepareStatement* and passing the parameterized SQL statement as an argument. Second, the client calls the *setString* method to assign a value to the first (and only) parameter. Third, the method calls *executeQuery* to execute the statement.

Figure 8-17 gives the API for the most common *PreparedStatement* methods. The methods *executeQuery* and *executeUpdate* are similar to the corresponding methods in *Statement*; the difference is that they do not require any arguments. The methods *setInt* and *setString* assign values to parameters. In Figure 8-16, the call to *setString* assigned a department name to the first index parameter. Note that the *setString* method automatically inserts the single quotes around its value, so that the client doesn't have to.

```
public ResultSet executeQuery()              throws SQLException;
public int       executeUpdate()             throws SQLException;
public void      setInt(int index, int val)  throws SQLException;
public void setString(int index, String val) throws SQLException;
```

Figure 8-17: Part of the API for *PreparedStatement*

Most people find it more convenient to use prepared statements to code dynamically-generated SQL statements than to create the SQL statement explicitly. Moreover, prepared statements are the more efficient option when statements are executed in a loop, as shown in Figure 8-18. The reason is that the database server is able to compile a prepared statement without knowing its parameter values. It therefore compiles the statement once, and then executes it repeatedly inside of the loop without further recompilation.

```
// Prepare the query
String qry = "select SName, GradYear "
           + "from STUDENT, DEPT "
           + "where DId = MajorId and DName = ?";
PreparedStatement pstmt = conn.prepareStatement(qry);

// Repeatedly get parameters and execute the query
String major = getUserInput();
while (major != null) {
   pstmt.setString(1, major);
   ResultSet rs = pstmt.executeQuery();
   displayResultSet(rs);
   major = getUserInput();
}
```

Figure 8-18: Using a prepared statement in a loop

8.2.5 Scrollable and updatable result sets

Result sets in basic JDBC are *forward-only* and *non-updatable*. Full JDBC also allows result sets to be *scrollable* and *updatable*. Clients can position such result sets at arbitrary records, update the current record, and insert new records. Figure 8-19 gives the API for these additional methods.

Methods used by scrollable result sets

```
   public void    beforeFirst()        throws SQLException;
   public void    afterLast()          throws SQLException;
   public boolean previous()           throws SQLException;
   public boolean next()               throws SQLException;
   public boolean absolute(int pos)    throws SQLException;
   public boolean relative(int offset) throws SQLException;
```

Methods used by updatable result sets

```
   public void updateInt(String fldname, int val)
                                       throws SQLException;
   public void updateString(String fldname, String val)
                                       throws SQLException;
   public void updateRow()             throws SQLException;
   public void deleteRow()             throws SQLException;
   public void moveToInsertRow()       throws SQLException;
   public void moveToCurrentRow()      throws SQLException;
```

Figure 8-19: Part of the API for *ResultSet*

The method *beforeFirst* positions the result set before the first record; that is, it resets the position to the same as when the result set was created. Similarly, the method *afterLast* positions the result set after the last record. The method *absolute* positions the result set at exactly the specified record, and returns *false* if there is no such record. The method *relative* positions the result set a relative number of rows. In particular, *relative(1)* is identical to *next*, and *relative(-1)* is identical to *previous*.

The methods *updateInt* and *updateString* modify the specified field of the current record on the client. However, the modification is not sent to the database until *updateRow* is called. The need to call *updateRow* is somewhat awkward, but it allows JDBC to batch updates to several fields of a record into a single call to the server.

Insertions are handled by the concept of an *insert row*. This row does not exist in the table (for example, you cannot scroll to it). Its purpose is to serve as a staging area for new records. The client calls *moveToInsertRow* to position itself at the insert row, then the *updateXXX* methods to set the values of its fields, then *updateRow* to insert the record into the database, and finally *moveToCurrentRow* to reposition the record set where it was before the insertion.

By default, record sets are forward-only and non-updatable. If a client wants a more powerful record set, it specifies so in the *createStatement* method of *Connection*. Basic JDBC contains only a no-arg *createStatement* method. But there is also a two-arg method, in which the client specifies scrollability and updatability. For example, consider the following statement:

```
Statement stmt =
         conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                              ResultSet.CONCUR_UPDATABLE);
```

All result sets that are generated from this statement will be scrollable and updatable. The constant TYPE_FORWARD_ONLY specifies a non-scrollable result set, and the constant CONCUR_READ_ONLY specified a non-updatable result set. These constants can be mixed and matched to obtain the desired scrollability and updatability.

For an example, recall the code of Figure 8-9, which allowed a user to iterate through the COURSE table, deleting desired records. Figure 8-20 revises that code to use updatable result sets. Note that a deleted row remains current until the call to *next*.

```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutocommit(false);

Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                      ResultSet.CONCUR_UPDATABLE);

ResultSet rs = stmt.executeQuery("select * from COURSE");
while (rs.next()) {
   String title = rs.getString("Title");
   boolean goodCourse = getUserDecision(title);
   if (!goodCourse)
      rs.deleteRow();
}
rs.close();
conn.commit();
```

Figure 8-20: Revising the code of Figure 8-9

A scrollable result set has limited use, because most of the time the client knows what it wants to do with the output records, and doesn't need to examine them twice. A client would typically need a scrollable result set only if it allowed users to interact with the result of a query. For example, consider a client that wants to display the output of a query as a Swing *JTable* object. The JTable will display a scrollbar when there are too many output records to fit on the screen, and allow the user to move back and forth through the records by clicking on the scrollbar. This situation requires the client to supply a scrollable resultset to the *JTable* object, so that it cn retrieve previous records when the user scrolls back.

8.2.6 Additional data types

In this chapter we have talked exclusively about manipulating integer and string values. JDBC also contains methods that manipulate numerous other types. For example, consider the interface *ResultSet*. In addition to the methods *getInt* and *getString*, there are also methods *getFloat*, *getDouble*, *getShort*, *getTime*, *getDate*, and several others. Each of these methods will read the value from the specified field of the current record, and convert it (if possible) to the specified Java type. In general, of course, it makes most sense to use numeric JDBC methods (such as *getInt*, *getFloat*, etc.) on numeric SQL fields, and so on. But JDBC will attempt to convert any SQL value to the Java type specified by the method. In particular, it is always possible to convert any SQL value to a Java string.

8.3 Computing in Java vs. SQL

Whenever a programmer writes a JDBC client, an important decision must be made:

- What part of the computation should be performed by the database server?
- What part of the computation should be performed by the Java client?

This section examines these questions.

Consider again the *StudentMajor* demo client of Figure 8-2. In that program, the server performs all of the computation, by executing an SQL query to compute the join of the STUDENT and DEPT tables. The client's only responsibility is to retrieve the query output and print it.

In contrast, we could have written the client so that it does all of the computation, as in Figure 8-21. In that code, the server's only responsibility is to download the entire STUDENT and DEPT tables. The client does all the rest of the work, computing the join and printing the result.

```
public class StudentMajor {
    public static void main(String[] args) {
      Connection conn = null;
      try {
         // Step 1: connect to database server
         Driver d = new ClientDriver();
         String url = "jdbc:derby://localhost/studentdb;create=false;";
         conn = d.connect(url, null);

         // Step 2: execute the queries
         Statement stmt1 = conn.createStatement();
         Statement stmt2 = conn.createStatement(
                                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                                    ResultSet.CONCUR_READ_ONLY);

         ResultSet rs1 = stmt1.executeQuery("select * from STUDENT");
         ResultSet rs2 = stmt2.executeQuery("select * from DEPT");

         // Step 3: loop through the result sets
         System.out.println("Name\tMajor");
         while (rs1.next()) {
            // get the next student
            String sname = rs.getString("SName");
            String dname = null;
            rs2.beforeFirst();
            while (rs2.next())
               // search for the major department of that student
               if (rs2.getInt("DId") == rs1.getInt("MajorId")) {
                  dname = rs2.getString("DName");
                  break;
               }
            System.out.println(sname + "\t" + dname);
         }
         rs1.close();
         rs2.close();
      }
      catch(SQLException e) {
         e.printStackTrace();
      }
      finally {
         // Step 4: Disconnect from the server
         try {
            if (conn != null)
               conn.close();
         }
         catch (SQLException e) {
            e.printStackTrace();
         }
      }
    }
}
```

Figure 8-21: An alternative (but bad) way to code the *StudentMajor* client

Which of these two versions is better? Clearly, the original version is more elegant. Not only does it have less code, but the code is easier to read. But what about efficiency?

As a rule of thumb, it is always more efficient to do as little as possible in the client. There are two main reasons:

- There is usually less data to transfer from server to client, which is important if they are on different machines.
- The server contains detailed specialized knowledge about how each table is implemented, and the possible ways to compute complex queries (such as joins). It is highly unlikely that a client can compute a query as fast as the server can.

For example, the code of Figure 8-21 computes the join by using two nested loops. The outer loop iterates through the STUDENT records. The inner loop runs for each student, and searches the DEPT table for the record matching that student's major.

Although this is a reasonable join algorithm, it is not particularly efficient. Part 4 of this book discusses several techniques that the server can use to compute the join much faster. For example, we saw in Chapter 6 that if DEPT has an index on *DId*, then the matching department for a given student could be found directly, and an inner loop would not be needed.

Figures 8-2 and 8-21 exemplify the extremes of really good and really bad JDBC code, and so comparing them was pretty easy. But sometimes the comparison is more difficult.

For example, consider again the *FindMajors* demo client of Figure 8-15, which returns the students having a specified major department. That code asks the server to execute an SQL query that joins STUDENT and MAJOR. We know that executing a join can be time-consuming. Suppose that after some serious thought, we realize that we can get the data we need without using a join. Our idea is to use two single-table queries. The first query scans through the DEPT table looking for the record having the specified major name; it then returns the *DId*-value of that record. The second query then uses that ID value to search the *MajorID* values of STUDENT records. The code for this algorithm appears in Figure 8-22. To save space, the figure shows only Step 2 of the entire program, similar to Figure 8-16.

```
      // Step 2: execute two single-table queries

      String qry1 = "select DId from DEPT where DName = ?";
      PreparedStatement pstmt1 = conn.prepareStatement(qry1);
      pstmt1.setString(1, major);
      ResultSet rs1 = pstmt1.executeQuery();
      rs1.next();
      int deptid = rs1.getInt("DId");  // get the specified major’s ID
      rs1.close();

      String qry2 = "select * from STUDENT where MajorId = ?";
      PreparedStatement pstmt2 = conn.prepareStatement(qry2);
      pstmt2.setString(1, deptid);
      ResultSet rs2 = pstmt2.executeQuery();
```

Figure 8-22: A clever way to recode Step 2 of the *FindMajors* client

This algorithm is simple, elegant, and efficient. It only requires a sequential scan through each of two tables, and ought to be much faster than a join. We can be proud of our effort.

Unfortunately, our effort is wasted. Our new algorithm isn’t really new, but just a clever implementation of a join (in particular, it is a *multi-buffer product* of Chapter 23 with a materialized inner table). A well-written database server would know about this algorithm (among several others), and would use it to compute the join if it turned out to be most efficient. All of our cleverness has thus been pre-empted by the database server. The moral is the same as with the *StudentMajor* client: Letting the server do the work tends to be the most efficient (as well as the easiest to code) strategy.

One of the mistakes that beginning JDBC programmers make is that they try to do too much in the client. The programmer might think that she knows a really clever way to implement a query in Java. Or the programmer might not be sure how to express a query in SQL, and feels more comfortable coding the query in Java. In each of these cases, the decision to code the query in Java is almost always wrong. The programmer must trust that the database server will do its job.[†]

8.4 Chapter Summary

- The JDBC methods manage the transfer of data between a Java client and a database server.

[†] At least, we should start by trusting that the server will be efficient. If we discover that our application is running slowly because the server is not executing the join efficiently, then we can recode the program as in Figure 8-22. But it is always best to avoid premature cleverness.

- Basic JDBC consists of five interfaces: *Driver*, *Connection*, *Statement*, *ResultSet*, and *ResultSetMetaData*. The following code fragment illustrates how their methods are used.

```
Driver d = new ClientDriver();
String url = "jdbc:derby://localhost/studentdb;create=false;";
Connection conn = d.connect(url, null);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select * from STUDENT");
while (rs.next())
      System.out.println(rs.getString("SName"));
rs.close();
conn.close();
```

- A *Driver* object encapsulates the low-level details for connecting with the server. If a client wants to connect to a server, it must obtain a copy of the appropriate driver class. In the above fragment, this class is *ClientDriver*, which is supplied by the Derby database.

- The driver class and its connection string are the only vendor-specific code in a JDBC program. Everything else refers to vendor-neutral JDBC interfaces.

- Result sets and connections hold resources that other clients might need. A JDBC client should always close them as soon as it can.

- Every JDBC method can throw an *SQLException*. A client is obligated to check for these exceptions.

- The methods of *ResultSetMetaData* provide information about the schema of the output table – that is, the name, type, and display size of each field. This information is useful when the client accepts queries directly from the user, as in an SQL interpreter.

- A basic JDBC client calls the driver class directly. Full JDBC provides the class *DriverManager* and the interface *DataSource* to simplify the connection process and make it more vendor-neutral.

- The class *DriverManager* holds a collection of drivers. A client registers its drivers with the driver manager, either explicitly or (preferably) via a system properties file. When the client wants to connect to a database, it provides a connection string to the driver manager and it makes the connection for the client.

- A *DataSource* object is even more vendor-neutral, because it encapsulates both the driver and the connection string. A client can therefore connect to a server without knowing any of the connection details. The database administrator can create various *DataSource* objects and place them on a server for clients to retrieve.

- A basic JDBC client ignores the existence of transactions. The database server executes these clients in autocommit mode, which means that each SQL statement is its own transaction.

- All of the database interactions in a transaction are treated as a unit. A transaction *commits* when its current unit of work has completed successfully. A transaction *rolls back* when it cannot commit. The database server implements a rollback by undoing all changes made by that transaction.

- Autocommit is a reasonable default mode for simple, unimportant JDBC clients. If a client performs critical tasks, then its programmer should carefully analyze its transactional needs.

- A client turns off autocommit by calling *setAutoCommit(false)*. This call causes the server to start a new transaction. The client then calls *commit* or *rollback* when it needs to complete the current transaction and begin a new one.

- When a client turns off autocommit, it must handle failed SQL statements by rolling back the associated transaction.

- A client can also use the method *setTransactionIsolation* to specify its isolation level. JDBC defines four isolation levels:
    - *Read-Uncommitted* isolation means no isolation at all. The transaction could have problems resulting from reading uncommitted data, nonrepeatable reads, or phantom records.
    - *Read-Committed* isolation forbids a transaction from accessing uncommitted values. Problems related to nonrepeatable reads and phantoms are still possible.
    - *Repeatable-Read* isolation extends read-committed so that reads are always repeatable. The only possible problems are due to phantoms.
    - *Serializable* isolation guarantees that no problems will ever occur.

- Serializable isolation is clearly to be preferred, but its implementation tends to cause transactions to run slowly. The programmer must analyze the risk of possible concurrency errors with the client, and choose a less restrictive isolation level only if the risk seems tolerable.

- A *prepared statement* has an associated SQL statement, which can have placeholders for parameters. The client can then assign values to the parameters at a later time, and then execute the statement.

- Prepared statements are a convenient way to handle dynamically-generated SQL statements. Moreover, a prepared statement can be compiled before its parameters are assigned, which means that executing a prepared statement multiple times (such as in a loop) will be very efficient.

- Full JDBC allows result sets to be *scrollable* and *updatable*. By default, record sets are forward-only and non-updatable. If a client wants a more powerful record set, it specifies so in the *createStatement* method of *Connection*.

- The rule of thumb when writing a JDBC client is to let the server do as much work as possible. Database servers are remarkably sophisticated, and usually know the most efficient way to obtain the desired data. It is almost always a good idea for the client to determine an SQL statement that retrieves exactly the desired data, and submit it to the server. In short, the programmer must trust the server to do its job.

8.5 Suggested Reading

A comprehensive and well-written book on JDBC is [Fisher et al. 2003], part of which exists as an online tutorial at *java.sun.com/docs/books/tutorial/jdbc*. Every database vendor supplies documentation explaining the use of its drivers, as well as other vendor-specific issues. If you intend to write clients for a specific server, then it is imperative to be familiar with the documentation.

8.6 Exercises

CONCEPTUAL EXERCISES

8.1 The Derby documentation recommends that you turn autocommit off when executing a sequence of inserts. Explain why you think it makes this recommendation.

PROGRAMMING EXERCISES

8.2 For each query in Exercises 4.20 and 4.21, write a program using Derby that executes the query and prints its output table.

8.3 The Derby *ij* client allows user input to comprise multiple lines, and ending with a semicolon. The *SQLInterperter* client, on the other hand, requires user input to be a single line of text. Revise *SQLInterpreter* so that user input can also comprise multiple lines and terminates with a semicolon.

8.4 In Section 7.5 we saw that the Derby *ij* client allows the user to create cursors and iterate through them. Revise the Derby version of the *SQLInterpreter* client so that it also supports cursor operations. (NOTE: Your code will need to use updatable result sets, which are not supported by the SimpleDB server.)

8.5 Write a class *SimpleDataSource* that implements the methods of *DataSource* mentioned in this chapter, and add it to the package *simpledb.remote*. (You don't need to implement all of the methods in *javax.sql.DataSource*.) What vendor-specific methods should it have?

8.6 It is often useful to be able to create a text file that contains SQL commands. These commands can then be executed in batch by a JDBC program.
a) Write a JDBC program that reads in commands from a specified text file and executes them. Assume that each line of the file is a separate command.
b) Do part (a), but assume that commands can span multiple lines and are separated by the semicolon character.

8.7 Investigate how a result set can be used to populate a Java *JTable* object. (HINT: You will need to extend the class *AbstractTableModel*.) Then revise the demo client *FindMajors* to have a GUI interface that displays its output in a JTable.

8.8 Write JDBC code for the following tasks:
c) Import data from a text file into an existing table. The text file should have one record per line, with each field separated by tabs. The first line of the file should be the names of the fields. The client should take the name of the file and the name of the table as input, and insert the records into the table.
d) Export data to a text file. The client should take the name of the file and the name of the table as input, and write the contents of each record into the file. The first line of the file should be the names of the fields.

8.9 This chapter has ignored the possibility of null values in a result set. To check for null values, you can use the method *wasNull* in *ResultSet*. Suppose you call *getInt* or *getString* to retrieve a field value. You can then call *wasNull* immediately afterwards to see if the retrieved value is null. For example, the following loop prints out graduation years, assuming that some of them might be null:

```
while(rs.next()) {
     int gradyr = rs.getInt("gradyear");
     if (rs.wasNull())
          System.out.println("null");
     else
          System.out.println(gradyr);
}
```

a) Rewrite the code for the *StudentMajor* demo client under the assumption that student names might be null.
b) Rewrite the code for the *SQLInterperter* demo client under the assumption that any field value might be null.

9

PERSISTENT JAVA OBJECTS

and the package javax.jpa

JDBC is the mechanism by which a Java program communicates with a database server. Consequently, a lot of the JDBC code in a program is concerned with low-level details that are irrelevant to the overall task of the program. Java programmers often write classes specifically to hide this JDBC code, in order to be able to handle data in a more object-oriented way. These classes constitute what is called the *object-relational mapping*. In this chapter we examine the requirements of a good object-relational mapping. We then introduce the Java Persistence Architecture, which makes it easy to create these mappings.

9.1 The Domain Model and View of a Client Program

9.1.1 The impedance mismatch

The purpose of JDBC is to facilitate the interaction between a Java program and a database server. JDBC contains methods to establish a connection with the server, methods to execute SQL statements, methods to iterate through the output records of a query, and methods to examine the values of the output records. The example clients of Chapter 8 nicely illustrated the use of these methods.

However, those example clients did not illustrate how difficult it is to write a good JDBC client. Those programs were just too small. As Java programs get larger, they need to be structured according to object-oriented principles. And the fundamental object-oriented principle is this:

> *An object-oriented program should manipulate objects, not values.*

Unfortunately, JDBC result sets do not follow this principle. When a JDBC client iterates through a result set, it extracts values from output rows. No conceptually meaningful objects are involved. For example, consider again the demo client *FindMajors* in Figure 8-15, which retrieves the STUDENT records having a user-specified major. Its *while*-loop extracts the *SName* and *GradYear* value from each output record of the result set, and prints them. In essence, the program is manipulating generic tables and records, not meaningful objects.

In contrast, consider the code fragment of Figure 9-1. This code does the same thing as *FindMajors*, but instead of using JDBC it uses a class called *DBStudent*. Each *DBStudent* object corresponds to a student in the database. That class has methods *getName* and *getGradYear*, which extract the appropriate field values from an object. The class also has the static method *findMajors* to return the collection of all *DBStudent* objects having a specified major.

```
String major = args[0];
System.out.println("Here are the " + major + " majors");
System.out.println("Name\tGradYear");

Collection<DBStudent> students = DBStudent.findMajors(major);
for (DBStudent s : students) {
      String sname = s.getName();
      int gradyear = s.getGradYear();
      System.out.println(sname + "\t" + gradyear);
}
```

Figure 9-1: A code fragment of more object-oriented version of *FindMajors*

The code of Figure 9-1 is easier to write and easier to understand than the equivalent JDBC code of Figure 8-15. The class *DBStudent* makes the programmer's job easy, because its methods are tailored specifically for the STUDENT table and it hides all of the messy interaction with the database server. In fact, the class *DBStudent* is absolutely perfect, except for one small thing – It is fictitious. Such a class might be fun to daydream about, but the reality is that we are stuck with JDBC.

The point is that there is a large difference between what the programmer wants from the database (i.e. *DBStudent* objects), and what the programmer gets from the database via JDBC (i.e. rows from the STUDENT table). This difference is inherently irreconcilable, because the database system is based on tables of records, whereas Java programs are based on classes of objects. This difference is commonly known as the *impedance mismatch*.[†]

The impedance mismatch is the irreconcilable difference between value-based database tables and object-based Java classes.

The standard way to deal with the impedance mismatch in this example is to implement the class *DBStudent* ourselves, using JDBC. It is reasonably easy to write a simple implementation of this class; see Exercise 9.5. For example, the code for its static method *findMajors* could create the SQL query string, execute the query to obtain a result set, loop through the result set to create a *DBStudent* object for each output record, and

[†] The term *impedance mismatch* comes from electronics, which recognized that circuits having mismatched impedances cannot be easily connected. It was first used in the context of databases by [Copeland and Maier 1984].

return a collection of those *DBStudent* objects. However, such an implementation turns out to have a variety of serious problems. In Section 9.2 we shall examine these problems, and consider ways to resolve them.

9.1.2 Separating the domain model from the view

Consider again the *FindMajors* client. Its functionality can be divided into two parts:

- The user interface (also called the *program view*), which reads the specified major from the command line and prints out the results.
- The *domain model*, which constructs the SQL query and accesses the database.

In fact, the functionality of every computer program can be divided into a view portion (which is responsible for accepting input and displaying output) and a model portion (which is responsible for performing calculations and manipulating data). The importance of this distinction arises from the following commonly-accepted design rule: *In any object-oriented program, the view portion and the model portion should be in disjoint classes*. This rule is usually expressed as follows:

An object-oriented program should separate the view from the model.

According to this rule, each program should contain view classes, which are concerned exclusively with the user interface. These classes know how to accept input from the user, but have no idea how it gets processed. Similarly, they know how to display data, but they have no idea about where that data comes from or what it means.

Similarly, each program should contain domain model classes, which are concerned exclusively with the state of the program. They know how to access the database, and have methods for storing and retrieving data. However, they have no idea which view is calling their methods, and do not care how the return values are displayed.

There are two advantages to separating the view from the domain model:

- *The code becomes easier to write*. Because the two parts of the program interact with each other in limited ways, each part can be written independently.
- *The code is more flexible*. Once the domain model has been written, it can be used in lots of different programs, each having a different view.

The situation is similar to that of a client-server system. Here, the view acts as the client and the domain model acts as the server, as it responds to each request made by the view.

The *FindMajors* code of Figure 9-1 is an example of code that separates the view from the model. The class *DBStudent* is the domain model, and the code fragment is the view. On the other hand, the *FindMajors* code of Figure 8-15 does not separate the view from the model, because the code to access the database is in the same method as (and intertwined with) the code to read the input and display the results.

In Chapter 1 we saw that the term *model* also has a database-specific meaning, which is related to the structure of its data (as in “the relational data model”). However, this meaning has nothing to do with the notion of a domain model, and is an unfortunate terminology conflict. Don't confuse them.

9.1.3 The object-relational mapping

The classes in a domain model can have several purposes. Some classes are concerned specifically with accessing the database; these classes form what is called the *data layer* of the model. Other classes support the needs of the business, such as performing data analysis, planning, input verification, and so on; these classes form the model's *business layer*.

Since this is a text on databases, our interest is in the data layer, and we shall focus on it exclusively. In fact, all of the examples in this chapter are so database-centric that their domain models do not even have a business layer! Issues surrounding the business layer are covered by books in object-oriented design, some of which are mentioned in Section 9.7.

The classes in the data layer are responsible for managing the connection to the database, knowing how and when to access the database, deciding when to commit and rollback transactions, and so on.

The core of the data layer consists of classes that correspond directly to the database tables. Typically each table has its own class, and each object of the class contains the data for one record. The class *DBStudent* in Figure 9-1 was an example of such a class. These classes are often called the *object-relational mapping* (or *ORM*), because they are responsible for mapping the database records to Java objects.

> *The object-relational mapping consists of the data-layer classes that correspond directly to the tables in the database.*

9.1.4 The ORM for the university database

In this section we examine the object-relational mapping for the university database. Consider the API of Figure 9-2.

Student

```
public int    getId();
public String getName();
public int    getGradYear();
public Dept   getMajor();
public Collection<Enroll> getEnrollments();
public void   changeGradYear(int newyear);
public void   changeMajor(Dept newmajor);
```

Dept

```
public int    getId();
public String getName();
public Collection<Student> getMajors();
public Collection<Course>  getCourses();
public void   changeName(String newname);
```

Enroll

```
public int     getId();
public String  getGrade();
public Student getStudent();
public Section getSection();
public void    changeGrade(String newgrade);
```

Section

```
public int    getId();
public String getProf();
public int    getYear();
public Course getCourse();
public Collection<Enroll> getEnrollments();
public void   changeProf(String newprof);
```

Course

```
public int    getId();
public String getTitle();
public Dept   getDept();
public Collection<Section> getSections();
public void   changeTitle(int newtitle);
```

Figure 9-2: The API for the object-relational mapping of the university database

Note how these classes correspond closely to the underlying database. The methods in each class allow an object to retrieve the field values in its record, and to change those field values that should be changed. For example, consider the class *Student*. Its methods *getId*, *getName*, and *getGradYear* retrieve the corresponding field values, and the method *changeGradYear* changes that field value. There is no method *changeId* or *changeName*, because the model assumes that students will not change their ID or their name.

Note also how these classes treat a table’s foreign key fields differently from its other fields. In particular, the classes do not have methods to return a foreign key value; instead, they contain methods to return the object pointed to by the foreign key. For example in the class *Student*, the method *getMajor* returns a *Dept* object, instead of the integer value of *MajorId*.

Because the classes do not hold foreign keys, the object-relational mapping looks a lot like a class diagram. For example, compare the API of Figure 9-2 with the class diagram of Figure 3-1 (minus the PERMIT class). The only thing that is missing from Figure 3-1 is an indication of the methods for each class.

The relationships between the classes in an object-relational mapping are implemented by methods. In particular, each class in an ORM will have a method for each relationship that it participates in. Thus there will be two methods for each relationship – one for each direction. We say that the relationships in the mapping are *bi-directional*.

As an example, consider the many-one relationship between the tables STUDENT and DEPT. The class *Student* contains the method *getMajor*, which returns the *Dept*-object corresponding to that student’s major department; and the class *Dept* contains the method *getMajors*, which returns the collection of those *Student*-objects who are majoring in that department. Similarly, the relationship between STUDENT and ENROLL is represented by two methods: The method *getStudent* in *Enroll*, and the method *getEnrollments* in *Student*. You should verify that every relationship in the class diagram of Figure 3-1 (except for the relationship between PERMIT and STUDENT) is represented by two methods in the API of Figure 9-3.

These methods allow a program to “navigate” through the relationships in the database. For example, consider the code fragment of Figure 9-3. The code starts with a single *Student* object, and uses the methods to find the student’s enrollments, the sections of these enrollments, the courses of those sections, and the titles of those courses. As with Figure 9-1, this code is quite elegant, and is much nicer than any analogous JDBC code.

```
Student s = ...  // somehow obtain a Student object
for (Enroll e : s.getEnrollments()) {
      Section k = e.getSection();
      Course  c = k.getCourse();
      String title = c.getTitle();
      System.out.println(title);
}
```

Figure 9-3: Printing the titles of the courses taken by a given student

9.1.5 The database manager

Figure 9-3 does not show how to obtain the initial *Student* object. There are two ways that you might want to obtain such an object:

- The object could correspond to an existing student, whose record is retrieved from the database.
- The object could correspond to a new student, whose record is to be inserted in the database.

The domain model needs to be able to handle each possibility.

To deal with an existing student, there needs to be a method to take a key value as its argument, find the corresponding record in the STUDENT table, and construct the appropriate *Student* object from that record. The method would return *null* if no student had that key value.

To deal with a new student, there needs to be a method to take the values of the new student as arguments, insert a new record into the STUDENT table, and return the corresponding *Student* object. The method would return *null* if a student already existed with that key value.

Note that neither of these methods can be implemented as a constructor, because there is no way that a constructor can return *null*. Instead, a common solution is to add another class to the data layer, which is to be responsible for these methods. Figure 9-4 gives the API for such a class, which we call *DatabaseManager*. Its methods are grouped into three sections, corresponding to the three responsibilities of the database manager.

```
// Methods to find existing database records by key value
public Student findStudent(int sid);
public Dept    findDept(int did);
public Enroll  findEnroll(int eid);
public Section findSection(int sectid);
public Course  findCourse(int cid);

// Methods to insert new records into the database
public Student insertStudent(int sid, String sname,
                             int gradyear, Dept major);
public Dept    insertDept(int did, String dname);
public Enroll  insertEnroll(int eid, Student student,
                            Section section);
public Section insertSection(int sectid, String prof,
                             int year, Course course);
public Course  insertCourse(int cid, String title, Dept dept);

// Methods to manage the connection to the database
public void commit();
public void close();
```

Figure 9-4: An API for the model class *DatabaseManager*

The database manager's first responsibility is to find records in the database by their key. Each of those methods searches the database for the specified record. If the record is found, the method constructs a model object corresponding to that record and returns it; otherwise, the method returns *null*.

The database manager's second responsibility is to insert records into the database. Each of those methods takes the appropriate values, inserts a record into the corresponding database table, constructs a model object having these values, and returns it. Note that those methods will need to extract the necessary foreign-key values from their arguments in order to do the insertion. For example, the last argument to the method *insertStudent* is a *Dept*-object. The code for that method will extract the ID-value from that object, and store it in the *MajorId* field of the new database record.

(By the way, note that the method *insertEnroll* does not have an argument for the grade. The intention is that new enrollments always have a default grade that indicates that the course is in progress; a final grade will be given at a later time, by calling the method *changeGrade*.)

The database manager's third responsibility is to manage the connection to the database. The connection is created by the *DatabaseManager* constructor, and closed by the method *close*. The method *commit* commits the current transaction and begins another one.

The code fragment of Figure 9-5 illustrates the use of the database manager to create two *Student* objects. The first object contains the data about student #1; the second object contains the data about a new student named Ron who has the same major as student #1.

```
DatabaseManager dbmgr = new DatabaseManager();
Student joe = dbmgr.findStudent(1);

Dept joesdept = joe.getMajor();
Student ron = dbmgr.insertStudent(12, "ron", 2009, joesdept);

dbmgr.commit();
dbmgr.close();
```

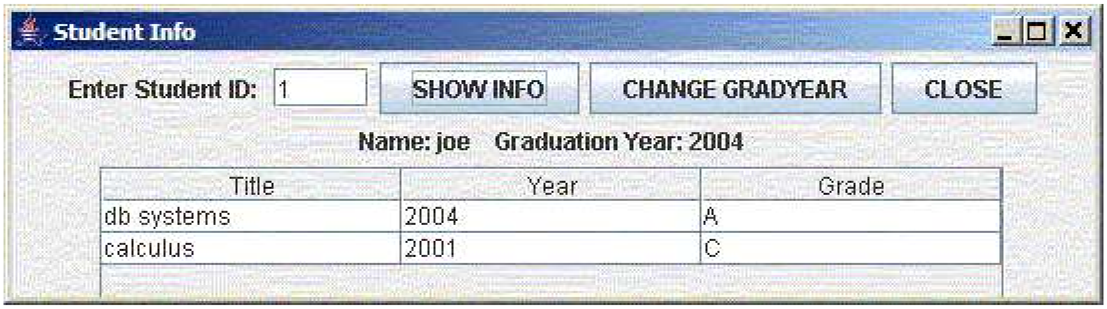
Figure 9-5: Using the class *DatabaseManager*

9.1.6 Building a view of the model

Now that we have an API for our domain model, we will need to implement its classes. However, writing this code is going to be a lot of work. Before we begin, we ought to be sure that the effort is going to be worthwhile.

Consider again the class *FindMajors* of Figure 8-15. This code would certainly be better if we rewrote it to use our domain model. But frankly, that example is not especially compelling, because the program is so small. The value of a domain model is more apparent in larger programs, such as clients that have a significant user interface. This section discusses such a client, called *StudentInfo*. This client is part of the SimpleDB distribution, in the *StudentInfo* subfolder of the Derby client demos.

When the *StudentInfo* client is run, it displays a screen that allows the user to perform two operations: to display information about the student having a specified id; and to change the graduation year of the student having a specified id. Figure 9-6 depicts the screen after a "1" was entered in the text box and the SHOW INFO button was pressed.



Enter Student ID: 1

Name: joe Graduation Year: 2004

Title	Year	Grade
db systems	2004	A
calculus	2001	C

Figure 9-6: A screenshot of the *StudentInfo* client

Pressing the CHANGE GRADYEAR button causes a dialog box to be displayed that asks for the new graduation year. After the user enters a value, the program changes the STUDENT record for the specified id, and displays the information for that student.

This code for this client appears in Figure 9-7. The code uses the Java Swing classes in a straightforward (and somewhat simplistic) way. What is of primary interest to us is how it interacts with the domain model. In fact, the view interacts with the model in just two places: in the method *main*, and in the button listeners.

```
public class StudentInfo {
   public static void main(String[] args) {
      DatabaseManager dbmgr = new DatabaseManager();
      JFrame frame = new TSFrame(dbmgr);
      frame.setVisible(true);
   }
}

class TSFrame extends JFrame {
   public TSFrame(DatabaseManager dbmgr) {
      setTitle("Student Info");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setSize(550,150);
      setLocation(200,200);
      getContentPane().add(new TSPanel(dbmgr));
   }
}

class TSPanel extends JPanel {
   private JLabel inputLbl  = new JLabel("Enter Student ID: ");
   private JTextField txt   = new JTextField(4);
   private JButton btn1     = new JButton("SHOW INFO");
   private JButton btn2     = new JButton("CHANGE GRADYEAR");
   private JButton btn3     = new JButton("CLOSE");
   private JLabel outputLbl = new JLabel("");
   private DefaultTableModel courses;

   public TSPanel(final DatabaseManager dbmgr) {
      Object[] columnNames = {"Title", "YearOffered", "Grade"};
      courses = new DefaultTableModel(columnNames, 0);
      JTable tbl = new JTable(courses);
      JScrollPane sp = new JScrollPane(tbl);
      add(inputLbl); add(txt); add(btn1); add(btn2); add(btn3);
      add(outputLbl); add(sp);

      btn1.addActionListener(
      new ActionListener() {
         public void actionPerformed(ActionEvent evt) {
            int sid = Integer.parseInt(txt.getText());
            Student s = dbmgr.findStudent(sid);
            display(s);
            dbmgr.commit();
         }
      });

   btn2.addActionListener(
      new ActionListener() {
         public void actionPerformed(ActionEvent evt) {
            String yearstring = JOptionPane.showInputDialog(
                                 "Enter new grad year");
            int sid  = Integer.parseInt(txt.getText());
            int newyear = Integer.parseInt(yearstring);
            Student s = dbmgr.findStudent(sid);
            s.changeGradYear(newyear);
            display(s);
            dbmgr.commit();
         }
      });

   btn3.addActionListener(
      new ActionListener() {
         public void actionPerformed(ActionEvent evt) {
            dbmgr.close();
            setVisible(false);
            System.exit(0);
         }
      });
   }

   private void display(Student s) {
      courses.setRowCount(0);
      if (s == null)
         outputLbl.setText("    No such student!");
      else {
         outputLbl.setText("Name: " + s.getName()
                       + " Graduation Year: " + s.getGradYear());
         for (Enroll e : s.getEnrollments()) {
            Section k = e.getSection();
            Course  c = k.getCourse();
            Object[] row = {c.getTitle(), k.getYear(),
                                  e.getGrade()};
            courses.addRow(row);
         }
      }
   }
}
```

Figure 9-7: The code for the *StudentInfo* view

The *main* method creates the database manager and passes it into the frame; likewise, the frame passes the database manager into its panel. This strategy is common practice – It ensures that only one database manager will get created, which will be shared by all parts of the view that need it. If the view happens to use multiple frames or panels, then they all will use the same database manager.

The action listeners of the buttons call the model's methods. In particular:

- The SHOW INFO listener asks the database manager to find the student having the specified ID. It then calls its private *display* method to output the information for that student, and commits.
- The CHANGE GRADYEAR listener uses a dialog box to request the new graduation year from the user. It then asks the database manager to find the specified student, and calls the method *changeGradYear* to modify the database. It then calls *display* to output the information for that student, and commits.
- The EXIT listener closes the database manager and quits.

The code for the method *display* is similar to the code in Figure 9-3. It navigates from the student to its collection of *Enroll* objects, the sections of those enrollments, and the courses of those sections. For each enrollment, it then extracts the title, year and grade from these objects and writes them to a row of the displayed *JTable*.

9.1.7 Implementing the domain model

Now that we have seen how easy it is to write a view for a model, we take up the task of implementing the model.

A popular implementation strategy is called the *Data Access Objects (DAO) pattern*. In this strategy, we implement each class of the object-relational mapping using two classes: a public class that contains the methods of the API; and a private class that contains the necessary methods to access the database. This private class is called the *DAO class*.

For example, consider our domain model for the university database, whose API appeared in Figure 9-2. The implementation of this model using the DAO pattern requires 11 classes: two classes for each of the five ORM classes, plus the class *DatabaseManager*. In this section we shall look at the code for three classes: *Student*, its associated DAO class *StudentDAO*, and the class *DatabaseManager*.

The class *Student*

Our implementation of the *Student* class appears in Figure 9-8.

```
public class Student {
   private StudentDAO dao;
   private int sid, gradyear;
   private String sname;
   private Dept major;
   private Collection<Enroll> enrollments = null;

   public Student(StudentDAO dao, int sid, String sname,
                  int gradyear, Dept major) {
      this.dao      = dao;
      this.sid      = sid;
      this.sname    = sname;
      this.gradyear = gradyear;
      this.major    = major;
   }

   public int getId() {
      return sid;
   }

   public String getName() {
      return sname;
   }

   public int getGradYear() {
      return gradyear;
   }

   public Collection<Enroll> getEnrollments() {
      if (enrollments == null)
         enrollments = dao.getEnrollments(sid);
      return enrollments;
   }

   public void changeGradYear(int newyear) {
      gradyear = newyear;
      dao.changeGradYear(sid, newyear);
   }

   public void changeMajor(Dept newmajor) {
      major = newmajor;
      dao.changeMajor(sid, newmajor);
   }
}
```

Figure 9-8: The code for the model class *Student*

Each *Student* object can be thought of as an in-memory version of its corresponding STUDENT record. It has local variables to hold the student’s id, name, graduation year, and major. These values are passed into the object when it is created. There are accessor methods to return these values, and mutator methods to change the student's graduation year and major.

When a mutator method gets called, the *Student* object needs to do two things. The first thing is to change the value of its local variable (i.e. either *gradyear* or *major*). The second thing is to update the database to correspond to this change. Since a *Student* object has no knowledge of the database, it calls its DAO object to execute the update.

The method *getEnrollments* works somewhat differently from the other methods. Each *Student* object contains a variable *enrollments*, which holds a collection of the *Enroll* objects for that student. However, a new *Student* object is not given a value for this variable; instead, the value of the *enrollments* variable is initially null. In fact, the variable stays null until the *getEnrollments* method is actually called. Only then does the *Student* object ask its DAO object to query the database and construct the collection. The *Student* object then saves this collection in its *enrollments* variable, and so subsequent calls to *getEnrollments* will not need to go to the database. This technique is called *lazy evaluation*, and will be discussed at length in the next section.

The class *StudentDAO*

The code for the class *StudentDAO* appears in Figure 9-9. The class contains five methods: *find*, *insert*, *getEnrollments*, *changeGradYear*, and *changeMajor*. The last three of these methods are called by *Student*, as we have seen. Each of those methods knows the appropriate SQL code to execute in order to retrieve or modify the desired data on the database.

```
public class StudentDAO {
   private Connection conn;
   private DatabaseManager dbm;

   public StudentDAO(Connection conn, DatabaseManager dbm) {
      this.conn = conn;
      this.dbm  = dbm;
   }

   public Student find(int sid) {
      try {
         String qry = "select SName, GradYear, MajorId "
                    + "from STUDENT where SId = ?";
         PreparedStatement pstmt = conn.prepareStatement(qry);
         pstmt.setInt(1, sid);
         ResultSet rs = pstmt.executeQuery();

         // return null if student doesn't exist
         if (!rs.next())
            return null;

         String sname = rs.getString("SName");
         int gradyear = rs.getInt("GradYear");
         int majorid  = rs.getInt("MajorId");
         rs.close();

         Dept major = dbm.findDept(majorid);
         return new Student(this, sid, sname, gradyear, major);
      }
      catch(SQLException e) {
         dbm.cleanup();
         throw new RuntimeException("error finding student", e);
      }
   }

   public Student insert(int sid, String sname,
                         int gradyear, Dept major) {
      try {
         // make sure that the sid is currently unused
         if (find(sid) != null)
            return null;

         String cmd = "insert into STUDENT(SId, SName, GradYear, "
                    + "MajorId) values(?, ?, ?, ?)";
         PreparedStatement pstmt = conn.prepareStatement(cmd);
         pstmt.setInt(1, sid);
         pstmt.setString(2, sname);
         pstmt.setInt(3, gradyear);
         pstmt.setInt(4, major.getId());
         pstmt.executeUpdate();
         return new Student(this, sid, sname, gradyear, major);
      }
      catch(SQLException e) {
         dbm.cleanup();
         throw new RuntimeException("error inserting new student", e);
      }
   }
   public Collection<Enroll> getEnrollments(int sid) {
      try {
         Collection<Enroll> enrollments = new ArrayList<Enroll>();
         String qry = "select EId from ENROLL where StudentId = ?";
         PreparedStatement pstmt = conn.prepareStatement(qry);
         pstmt.setInt(1, sid);
         ResultSet rs = pstmt.executeQuery();
         while (rs.next()) {
            int eid = rs.getInt("EId");
            enrollments.add(dbm.findEnroll(eid));
         }
         rs.close();
         return enrollments;
      }
      catch(SQLException e) {
         dbm.cleanup();
         throw new RuntimeException("error getting enrollments", e);
      }
   }

   public void changeGradYear(int sid, int newyear) {
      try {
         String cmd = "update STUDENT set GradYear = ? where SId = ?";
         PreparedStatement pstmt = conn.prepareStatement(cmd);
         pstmt.setInt(1, newyear);
         pstmt.setInt(2, sid);
         pstmt.executeUpdate();
      }
      catch(SQLException e) {
         dbm.cleanup();
         throw new RuntimeException("error changing grad year", e);
      }
   }

   public void changeMajor(int sid, Dept newmajor) {
      try {
         String cmd = "update STUDENT set MajorId = ? where SId = ?";
         PreparedStatement pstmt = conn.prepareStatement(cmd);
         pstmt.setInt(1, newmajor.getId());
         pstmt.setInt(2, sid);
         pstmt.executeUpdate();
      }
      catch(SQLException e) {
         dbm.cleanup();
         throw new RuntimeException("error changing major", e);
      }
   }
}
```

Figure 9-9: The code for the DAO class *StudentDAO*

The database manager calls the methods *find* and *insert*. Each of these methods creates a new *Student* object. The *find* method retrieves the STUDENT record having the specified key, and creates a *Student* object having those values. In order to create that object, the method needs to turn the foreign key *majorid* into the corresponding *Dept* object; it does so by calling the *findDept* method of the database manager. The *insert* method creates a *Student* object having the specified values. It also inserts a new record into the STUDENT table. Note how the method extracts the foreign key from the supplied *Dept* object, in order to store it in the new record.

The *find* and *insert* methods pass a reference to the *StudentDAO* object into each *Student* object that they create. This technique allows each *Student* object to be oblivious to the database; when a database action needs to occur, the *Student* object just asks its DAO object to do the work.

Each *StudentDAO* object contains a reference to the database manager. It uses this reference when it needs objects from other domain classes (such as *Dept* objects in the *find* method, and *Enroll* objects in the *getEnrollments* method). It also uses the reference to call the database manager's *cleanup* method when it handles an SQL exception.

The class *DatabaseManager*

The *DatabaseManager* class implements the API of Figure 9-4; its code appears in Figure 9-10. As we have seen, this class contains two methods for each ORM class; for example, the methods for *Student* are *findStudent* and *insertStudent*. The class implements these methods by simply calling the corresponding method of the appropriate DAO class.

```
public class DatabaseManager {
   private Connection conn;
   private StudentDAO studentDAO;
   private DeptDAO    deptDAO;
   private EnrollDAO  enrollDAO;
   private SectionDAO sectionDAO;
   private CourseDAO  courseDAO;

   public DatabaseManager() {
      try {
         Driver d = new ClientDriver();
         String url = "jdbc:derby://localhost/studentdb;create=false;";
         conn = d.connect(url, null);
         conn.setAutoCommit(false);

         studentDAO = new StudentDAO(conn, this);
         deptDAO    = new DeptDAO(conn, this);
         enrollDAO  = new EnrollDAO(conn, this);
         sectionDAO = new SectionDAO(conn, this);
         courseDAO  = new CourseDAO(conn, this);
      }
      catch(SQLException e) {
         throw new RuntimeException("cannot connect to database", e);
      }
   }

   public void commit() {
      try {
         conn.commit();
      }
      catch(SQLException e) {
         throw new RuntimeException("cannot commit database", e);
      }
   }

   public void close() {
      try {
         conn.close();
      }
      catch(SQLException e) {
         throw new RuntimeException("cannot close database", e);
      }
   }

   public Student findStudent(int sid) {
      return studentDAO.find(sid);
   }

   public Student insertStudent(int sid, String sname,
                                int gradyear, Dept major) {
      return studentDAO.insert(sid, sname, gradyear, major);
   }

//  the find and insert methods for Dept, Enroll, Section and Course
//  are similar, and thus are omitted here

   public void cleanup() {
      try {
         conn.rollback();
         conn.close();
      }
      catch(SQLException e) {
         System.out.println("fatal error: cannot cleanup connection");
      }
   }
}
```

Figure 9-10: The code for the class *DatabaseManager*

In object-oriented terminology, *DatabaseManager* is called a *façade class* for the DAO objects. It creates a DAO object for each domain class, and uses them to do its work. In this way, a client only needs to interact with a single object, instead of several individual DAO objects.

9.1.8 Lazy and Eager Evaluation

Each *Student* object stores its state in its private field variables. A *Student* object has one variable for each field of the STUDENT table, as well as a variable for each relationship. In particular, the variable *major* corresponds to the relationship with *Dept*, and the variable *enrollments* corresponds to the relationship with *Enroll*. The variable *major* holds a *Dept* object and the variable *enrollments* holds a collection of *Enroll* objects.

Note that the values for *major* and *enrollments* are computed at different times. The value for *major* is computed by the *find* method of *StudentDAO*, and is passed into the constructor for *Student*. On the other hand, the value of *enrollments* is initially *null*, and is computed only when *getEnrollments* is first called. The computation of *major* is known as *eager evaluation*, because the information is retrieved from the database before it is requested. Conversely, the computation of *enrollments* is known as *lazy evaluation*.

> *In eager evaluation, an object is computed before it is used.*
> *In lazy evaluation, an object is not computed until it is needed.*

Eager evaluation may seem strange at first – after all, why would you want to create an object before you need it? But eager evaluation can be a very efficient way to interact with the database. The reason is that it is much better to make a single call to the database that grabs a lot of data, than to make several calls to the database, each of which returns a small amount of data. So if we expect to do a lot of navigation (for example, from a *Student* object to its major department), then it is better to use eager evaluation to retrieve the data for both the *Student* and the *Dept* objects at the same time[†].

Lazy evaluation keeps the code from creating objects unnecessarily. For example, suppose that a new *Student* object gets created. If the variable *enrollments* is computed using eager evaluation, then the code will fetch that student's enrollment information from the database. If the user never looks at these *Enroll* objects, then the model would have executed a query unnecessarily.

Since the domain model cannot predict how its objects will be used, it must guess whether to use lazy or eager evaluation. Our implementation of the model takes a common approach, which is to do eager evaluation on forward relationships and lazy evaluation on inverse relationships.

One advantage to this approach is that it keeps the code from creating too many objects. Assume that we had chosen to compute *enrollments* using eager evaluation. Then when a new *Student* object gets created, the student's *Enroll* objects also get created. But the object creation doesn't stop there. When an *Enroll* object gets created, its associated *Section* object will also get created eagerly. And if *Section* also implements its *enrollments* object eagerly, then that will cause all of the *Enroll* objects for that section to get created, which will cause the *Student* objects for all students in that section to get created. In other words, creating one *Student* object will cause *Student* objects to get created for those students in the same sections as the original student. And then the *Enroll* objects for those students must get created, and so on. In fact, in the absence of lazy evaluation, creating one object will likely cause objects for the entire database to get created.

[†] You may have noticed that the method *find* in Figure 9-9 doesn't live up to this claim. It uses two separate queries to retrieve the data from STUDENT and from DEPT, thereby eliminating the advantage of eager evaluation. Exercise 9.6(e) asks to you remedy this problem.

9.2 The Problem With Our Domain Model

In the previous section we designed and developed a comprehensive domain model for our university database. It took us a while to develop the model, but the effort seems to be worth it. If we want to write a client that uses the database, all we have to do now is write its view. We don't even need to know SQL or JDBC, because all of the necessary code is encapsulated in the methods of the DAO classes. It's hard to imagine how things could be much better.

Unfortunately, our domain model is not very usable, because its implementation is very inefficient. There are several reasons for the inefficiency:

- Objects are not cached.
- Updates are performed immediately.
- The connection is continuously kept open until the program exits.
- There is no good way to find domain objects, except by their key.

The following subsections discuss these issues.

9.2.1 Caching objects

In our implementation, the DAO objects do not keep track of the objects that they create. Every time a *find* method is called, its DAO object goes to the database, executes a query, and constructs the object. It doesn't matter if that object has already been created; the DAO object always goes to the database.

Clearly, the DAO objects are being inefficient when they go to the database unnecessarily, because they have to wait for the database to return the desired information. Moreover, they are also wasting the time of the database, which could be serving other programs instead.

Duplicate object creation shows up in various guises. Consider the *StudentInfo* client. An obvious example of duplication occurs when a user explicitly requests a student's information more than once. A more subtle example occurs when the user requests the information for two students who happen to be in the same section; in this case, the DAO objects will create that *Section* object twice, as well as the related *Course* and *Dept* objects (due to eager evaluation).

A particularly nasty example of duplicate object creation can occur if eager evaluation is overused. Suppose that the enrollments for a student are computed eagerly. Then when a *Student* object is created, the corresponding *Enroll* objects will also get created. Each of these *Enroll* objects will then create their associated *Student* object, which ought to be the original student. But because objects are not cached, a duplicate *Student* object gets created. And this object then creates its *Enroll* objects, which are duplicates of the previous *Enroll* objects. And so on, in an infinite loop of object creation.

This infinite loop can also occur if there is a loop of forward relationships in the class diagram. For example, suppose that the every department has a work-study student that helps out in the office. Then when a *Student* object is created, it will create the *Dept* object denoting the student's major, which will create the *Student* object denoting the department's work-study student, which will create the *Dept* object for that student's major, and so on. As before, eager evaluation, without caching, leads to an infinite loop.

The solution to this problem is this:

> *Each DAO object should keep a cache*
> *of the objects that it has already created.*

The cache could be implemented as a map, keyed by the object's ID. The *find* method would look up the given ID in the map. If an object was found, it would be used; otherwise, the object would be created by going to the database.

9.2.2 Postponing database updates

Consider the model methods that insert new objects in the database or modify existing objects. Each of these methods performs two actions:

- It updates the object's internal state in order to be consistent with the change to the database.
- It executes an SQL query to update the database.

If a DAO object has been caching its objects, then it does not need to update the database immediately. The reason is that the updated objects will be in its cache, and thus the DAO object will never query the database for their values. Instead, the DAO object can choose to perform the updates later, such as when the transaction commits.

> *The DAO objects should postpone database updates*
> *for as long as possible.*

By postponing database updates, the DAO object can reduce the number of times it has to contact the database server, which improves its efficiency. For example, suppose that a client changes the value of several fields in an object. Instead of issuing an SQL update statement for each field, the DAO object could combine these modifications into a single SQL statement. Or for another example, suppose that the client performs a rollback. If the database updates have been postponed, then they can just be ignored. Since the database system was not told about the updates, it will not need to roll them back.

9.2.3 Managing connections

Database servers cannot support an unlimited number of connections. Each connection to a client requires server-side resources – For example, the server has to spawn a thread to handle the connection, and that thread needs to be scheduled. Allowing too many simultaneous connections degrades a server's ability to respond to requests. A database server provider typically charges its customers by the number of simultaneous connections it can have; a customer that uses many connections will have to pay more than a customer that uses just a few.

This situation means that clients should close connections when they are not needed. Our database manager is not at all careful with its connection – The manager opens a connection to the database in its constructor, and keeps the connection open until the manager is closed. Consider the consequences of this implementation on the *StudentInfo* client. A user in the Dean's office may open the application once at the beginning of the day, click on the SHOW INFO button periodically during the day, and then close the program when it is time to go home. This user is not aware of the fact that the program is hogging a valuable connection to the server; in fact, the user shouldn't be aware of this. Instead, it is the responsibility of the model to manage the connection more intelligently.

The basic principle is that the model must keep the connection to the database open during the execution of a transaction, but can choose to close it after the transaction commits. The connection can then be re-opened when the next transaction begins.

The model should be able to close the connection to the database between transactions.

Here is how to revise the code for *DatabaseManager* to implement this principle. The database manager does not obtain the connection in its constructor. Instead, we write a new method, called *begin*, which obtains the connection, and we modify the *commit* method so that it closes the connection. We then force clients to call *begin* each time that they begin a new transaction. Consequently, the database manager will open a connection at the beginning of a transaction, and close the connection at the end of the transaction.

Note that this new method *begin* changes how clients relate to the database manager. In JDBC, a new transaction is created immediately after the current transaction is committed; there is never a moment when a transaction is not running. This new method has changed that situation. Committing a transaction leaves the program in a "non-transactional" state; a new transaction will not start until the program calls the *begin* method. During this time between transactions the client can do non-database work, or as is the case in *StudentInfo*, sit idle.

This ability to be in a non-transactional state makes a lot of sense for our *StudentInfo* client. Each button press corresponds to a transaction. The button listener can begin the transaction, perform the appropriate actions, and then commit. The model is then free to close the database connection while the view waits for the user to click a button again.

The problem with repeatedly opening and closing connections is that these operations are time-consuming. It can take several seconds to contact a database server and log in. Our revision to the database manager would make the *StudentInfo* program become sluggish, because each button press would take much longer to complete. If our program needed to respond quickly to frequent button pushes, then we would be in trouble.

A common solution to this problem is to run a server, called a *connection pool*, that shares database connections among multiple clients. The connection pool begins by opening several connections to the database. Client programs request connections from the connection pool when they need them, and return them when they are done. The connection pool keeps track of which connections are currently in use. If all connections are currently used, then a requesting client will be placed on a wait list until a connection becomes available. Connection pools are especially useful in a web server environment, and we shall discuss them in more detail in Chapter 11.

9.2.4 Finding objects

The *findStudent* method in *DatabaseManager* can create a *Student* object if we give it the student's ID value. But what if we want to find students in some other way, such as the set of students graduating in a particular year? There are two approaches we could take, neither of which is very satisfactory.

The first approach is to add the method *findAllStudents* to the *DatabaseManager* class; calling this method would return a collection containing all students in the database. This method is a general-purpose method. We can obtain any subset of students that we want, by simply calling *findAllStudents* and then looping through the resulting objects. In fact, the problem with this approach is due to the general nature of *findAllStudents*: It forces the client to retrieve and examine every *Student* object, instead of letting the server do the work.

The second approach is to add the method *findStudentsByGradYear* to the *DatabaseManager* class. This method would be more efficient than *findAllStudents*, because it lets the server do the work. However, its problem is due to its specificity: Out of all of the possible specific-purpose methods, which do we add to *DatabaseManager*? Ideally, we would like our domain model's API to be stable. That is, we want the model to contain all of the necessary methods when we build it; we don't want to keep adding new methods over time. But we often don't know in advance what sets of objects a view might want.

9.3 The Java Persistence Architecture

The previous section discussed several deficiencies in our implementation of a domain model for the university database. None of these deficiencies is insurmountable. With a little bit of work (or more likely, a lot of work), we could rewrite the model's code so that all of these problems are resolved.

Of course, the university database is not the only database in the world. We might be asked to write domain models for other, possibly more extensive, databases. We really don't want to have to completely redo our work every time we write a new model.

Fortunately, a lot of the code will be the same in each model. For example, the code to support object caching, postponed updates, and connection management is independent of the database schema, and thus can be shared among the models. Moreover, the code to implement the find, insert, and change methods is basically the same for each model class, as are the SQL statements that those methods use. If we are clever, we could encapsulate almost all of the necessary code into a single package. The code for each individual domain model would then be simple and short, because it could import the contents of that common package.

In fact, this encapsulation has already been done. The code is known as the *Java Persistence Architecture* (or JPA), and is defined in the package *javax.jpa*. This section introduces the various JPA components, and shows how they are used to build a domain model. JPA uses some specific terminology, which is defined in Figure 9-11.

JPA Term	**Definition**
Persistence Unit	The classes comprising the object-relational mapping.
Entity	An object from a class in the persistence unit.
Entity Manager	An object that manages the entities from a specified persistence unit.
Persistence Context	The cache of entities managed by the entity manager.

Figure 9-11: Terminology used by JPA

9.3.1 The entity manager

The entity manager is the most important part of JPA. Entity managers implement the interface *EntityManager*. This interface is similar to our model class *DatabaseManager*, but it has more functionality and is domain independent. Figure 9-12 gives the API for five important *EntityManager* methods.

```
EntityTransaction getTransaction();
<T> T             find(Class<T> entityclass, Object key);
void              persist(Object entity);
void              remove(Object entity);
void              close();
```

Figure 9-12: Part of the API for the *EntityManager* interface

Figure 9-13 shows a code fragment that illustrates the use of these methods. The code is divided into six parts; the remainder of this section discusses these parts.

```
// Part 1:  Obtain the entity manager
EntityManagerFactory emf =
                Persistence.createEntityManagerFactory("studentdb");
EntityManager em = emf.createEntityManager();

// Part 2: Begin a transaction
EntityTransaction tx = em.getTransaction();
tx.begin();

// Part 3: Find an existing entity by its key
Dept mathdept = em.find(Dept.class, 20);

// Part 4: Insert a new entity
Student s = new Student(12, "ron", 2010, mathdept);
em.persist(s);

// Part 5: Modify and delete existing entities
s.changeGradYear(2011);
for (Enroll e : s.getEnrollments())
   em.remove(e);

// Part 6:  Deal with rollback exceptions
try {
   tx.commit();
}
catch (RollbackException ex) {
   ex.printStackTrace();
   tx.rollback();
}
finally {
   em.close();
}
```

Figure 9-13: A JPA client to test the entity manager

Obtaining an Entity Manager

Part 1 of Figure 9-13 shows how a JPA client obtains an entity manager. Two steps are required:

- Create an *EntityManagerFactory* object.
- Use the factory object to create an *EntityManager* object.

The JPA class *Persistence* has a static method *createEntityManagerFactory*. A client creates its entity manager factory by calling this method and passing it the name of the desired persistence unit. Persistence units are specified in a separate XML file, which will be discussed in Section 9.3.3. The code of Figure 9-13 assumes that a persistence unit named “studentdb” exists.

Once the client has the entity manager factory, it obtains the entity manager by calling the factory’s *createEntityManager* method. At first glance, this two-step process seems unnecessary. What's the point of having an entity factory? Why doesn't the *Persistence* class simply have the method *createEntityManager*?

The reason has to do with the fact that JPA can be used in different contexts, such as in a stand-alone application or with a web server. The use of a factory object allows the same code to be run in each context, the only difference being how the factory object is obtained. So although this code seems convoluted, its flexibility turns out to be extremely practical in the real world.

Dealing with Transactions

Part 3 of Figure 9-13 illustrates the use of transactions. The method *getTransaction* returns an object that implements the interface *EntityTransaction*. The important methods of that interface are *begin*, *commit*, and *rollback*.

The method *begin* starts a new transaction, and methods *commit* and *rollback* complete a transaction. Unlike in JDBC, a commit or rollback does not automatically begin a new transaction; instead, the client must explicitly call *begin*. As we mentioned in Section 9.2.3, this feature gives the entity manager the ability to close the database connection between transactions, if it so chooses.

Finding Existing Entities

Part 4 of Figure 9-13 illustrates how to use the *find* method. The method takes two arguments: the class of the desired entity; and the entity's key value. It then creates an entity for that class corresponding to the database record having that key. The method is similar to the methods *findDept*, *findStudent,* etc, of our class *DatabaseManager*; the difference is that the JPA *find* method is generic, and works with any class of the persistence unit.

The entity manager saves each entity it creates in its persistence context, and it performs caching on these entities. That is, if the entity manager is asked to find an entity, it first looks in the persistence context. If the entity is there, then it is returned to the client. Otherwise the entity manager accesses the database, creates the entity having the retrieved values, and saves the entity in its persistence context.

Inserting New Entities

Part 5 of Figure 9-13 illustrates how new entities are created. A client must use two statements to create an entity. First, the client creates a model object, using its constructor. Then the client calls the entity manager's method *persist*, which adds the object to the persistence context.

Note that this approach is different from how a client performs insertions in our domain model. In our model, a client would never call the *Student* constructor directly; instead, the client would obtain a *Student* object either by calling *findStudent* or *insertStudent*. In

JPA, however, clients are allowed to create their own domain objects. The catch is that such objects are not persistent, and are not associated with the database. An object becomes persistent only after the client passes it into the *persist* method.

This two-step approach provides for more flexibility than in our model. For example, a client can have several *Student* objects, some of which are persistent and some of which are not. This added flexibility, of course, can lead to added complexity – A client must be very careful to keep track of which objects have been persisted.

Once an object is in the persistence context, the entity manager will eventually insert a new record for it into the database via an SQL *insert* statement. However, the entity manager may choose to postpone the insertion for as long as possible. Typically, the new record will be inserted when the transaction commits. Because of caching, however, the client will not know or care when the insertion takes place.

Modifying and Deleting Existing Entities

Part 6 of Figure 9-13 illustrates how a client modifies and deletes an existing entity.

To modify an entity, the client simply calls a mutator method directly on the entity. The mutator method changes the state of the object, but does not immediately change the database. Instead, the entity manager notices the change to its persistence context, and marks that entity for update. Eventually, the entity manager will decide to synchronize the modified entities with the database. However, as with insertions, the entity manager can choose to postpone this update for as long as possible.

To delete an entity, the client calls the entity manager's *remove* method. The entity manager immediately removes the entity from the persistence context; it also makes note of the removal, so that it can eventually delete the corresponding record from the database.

Handling Exceptions

One notable feature of the entity manager is that its methods do not throw checked exceptions (unlike JDBC's *SQLException*). Instead, each JPA method throws the runtime exception *PersistenceException* or one of its subclasses.

The one JPA exception class that a client must catch is *RollbackException*. This exception is thrown when the *commit* method fails. The client is responsible for catching this exception and calling the *rollback* method to roll back the transaction.

The *finally* clause in Figure 9-13 ensures that the entity manager will be closed, even if an exception occurs (and regardless of whether the exception is caught).

9.3.2 Implementing the object-relational mapping

The JPA class *EntityManager* encompasses the functionality of our *DatabaseManager* class and all of our DAO classes. The entity manager takes care of all of the interaction with the database: Its *find* method retrieves from the database all of the data needed to construct a new entity; and its *persist* method inserts into the database the data it extracted from the specified entity. The classes in the object-relational mapping do not get involved with the database at all.

In our implementation of the domain model, the DAO classes contained the SQL statements for interacting with the database. The entity manager needs to be able to construct these SQL statements automatically for each class in the ORM. It therefore needs to know the following information about that class:

- Which database table does the class correspond to?
- Which class variables correspond to the fields of the record?
- Which field(s) form the key of the table?
- Which field(s) form the foreign key of each relationship?

This information is specified in the Java code for each class. Figure 9-14 illustrates the code for the *Student* class.

```
@Entity
@Table(name="STUDENT")
public class Student {

   @Id
   @Column(name="SId")
   private int sid;

   @Column(name="SName")
   private String sname;

   @Column(name="GradYear")
   private int gradyear;

   @ManyToOne
   @JoinColumn(name="MajorId")
   private Dept major;

   @OneToMany(mappedBy="student")
   private Collection<Enroll> enrollments;

   // JPA requires a no-arg constructor
   public Student() {}

   public Student(int sid, String sname, int gradyear, Dept major) {
      this.sid = sid;
      this.sname = sname;
      this.gradyear = gradyear;
      this.major = major;
   }

   public int getId() {
      return sid;
   }

   public String getName() {
      return sname;
   }

   public int getGradYear() {
      return gradyear;
   }

   public Dept getMajor() {
      return major;
   }

   public void changeGradYear(int year) {
      gradyear = year;
   }

   public void changeMajor(Dept dept) {
      major = dept;
   }

   public Collection<Enroll> getEnrollments() {
      return enrollments;
   }
}
```

Figure 9-14: The JPA version of the *Student* model class

This class is remarkably similar to the class *Student* of our model implementation, which appeared in Figure 9-8. The most obvious difference is the presence of annotations. (An annotation is an expression beginning with “@”.) These annotations are attached to the class and its variables, and tell the entity manager how the class corresponds to the database. We shall explain the annotations in a bit.

Another difference is in the code for the mutator methods and the lazily-evaluated accessor methods. Note how the JPA code is concerned only with the local state. The methods *changeGradYear* and *changeMajor* only change the local variable; they do not also call a method to update the database. And the code for *getEnrollments* is even stranger, because there is no code anywhere in the class to create the collection! In each of these cases, the entity manager handles these details implicitly.

We now consider the annotations used by JPA. There are two annotations attached to the class *Student*:

- The *@Entity* annotation specifies that the class maps to a table in the database.
- The *@Table* annotation specifies the name of the database table associated with the class.

Some of the variables in *Student* have a *@Column* annotation. These variables correspond to a field in the STUDENT table, and the annotation specifies this correspondence. The variable *sid* also has a *@Id* annotation; this annotation specifies that the corresponding field is the key of STUDENT.

There is a *@ManyToOne* annotation attached to each variable that denotes a foreign-key relationship. The annotation contains the name of the foreign key field in the database. For example in *Student*, the variable *major* is associated with the foreign key field *MajorId*. Note that this annotation gives the entity manager everything it needs to know about the relationship to create the necessary SQL query. For example, the variable *major* is of type *Dept*, which means that the relationship is between the STUDENT table and the table associated with the *Dept* class. Moreover, the entity manager knows that the predicate joining these two tables will equate the foreign key field (i.e. *MajorId*) with the key of *Dept*'s table.

There is a *@OneToMany* annotation attached to each variable that denotes an inverse relationship. Consider the variable *enrollments* in *Student*. This variable has type *Collection<Enroll>*, which means that it denotes the inverse of some relationship in the *Enroll* class. Which relationship? The expression *mappedBy="student"* inside the annotation tells us. In particular, it specifies that the relationship is the one held by the variable *student* in *Enroll*.

JPA annotations are a remarkably simple way to specify the mapping between a class and its database table. However, we can do even better: We can omit the *@Table* and *@Column* annotations if we name our class and fields appropriately. That is, the entity manager assumes that if the class does not have a *@Table* annotation, then the name of the database table is the same as the name of the class. Similarly, if a variable does not have a *@Column* annotation, then the name of its associated field is assumed to be the name of the variable. In Figure 9-14, all of the table and field names satisfy this naming convention[†]. Thus the *@Table* and all of the *@Column* annotations could be omitted from that code.

9.3.3 Persistence Descriptors

Recall that a JPA client obtains its entity manager via a two-step sequence. For example, Figure 9-13 had the following statements:

```
EntityManagerFactory emf =
                Persistence.createEntityManagerFactory("studentdb");
EntityManager em = emf.createEntityManager();
```

The only argument to the method *createEntityManagerFactory* is a string denoting the name of a persistence unit. This is not a lot of information. What the *createEntityManagerFactory* method really needs to know are details such as:

- Where is the Java class that implements the entity manager?
- What classes are in the object-relational mapping?
- Where is the database, and how should the entity manager connect to it?

[†] Recall that table names and field names are not case sensitive in SQL, so it doesn't matter if the Java names have different capitalization than in the database.

This information is stored in an XML file called the *persistence descriptor*. Figure 9-15 gives the persistence descriptor used for the *JPAStudent* client.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
                                                   version="1.0">
  <persistence-unit name="studentdb">
     <provider>
            org.apache.openjpa.persistence.PersistenceProviderImpl
     </provider>

     <class>Student</class>
     <class>Permit</class>
     <class>Enroll</class>
     <class>Section</class>
     <class>Course</class>
     <class>Dept</class>

     <properties>
       <property name="openjpa.ConnectionURL"
            value="jdbc:derby://localhost/ studentdb;create=false;"/>
       <property name="openjpa.ConnectionDriverName"
            value="org.apache.derby.jdbc.ClientDriver"/>
       <property name="openjpa.ConnectionUserName"
            value="APP"/>
       <property name="openjpa.ConnectionPassword"
            value="APP"/>
     </properties>
   </persistence-unit>
</persistence>
```

Figure 9-15: A typical JPA persistence descriptor

The persistence unit has three parts: *provider* tag, a set of *class* tags, and a *properties* tag. These tags must be in the order shown. Each tag addresses one of the above bullet points.

The *provider* tag specifies the vendor-supplied class that implements the entity manager. Here, the class is *PersistenceProviderImpl*, which (as you can tell from its package name) is supplied by Apache as part of its JPA implementation called *OpenJPA*.

The *class* tags list the classes in the object-relational mapping.

The *properties* tag provides information needed by the entity manager. The names of these properties are vendor-specific, and each vendor can choose which properties it wants to support. Figure 9-15 shows the two required properties for the OpenJPA implementation: *ConnectionURL* and *ConnectionDriver*. The value of these properties should look familiar – they are just the name of the database driver and its connection string. If the database requires you to log in, then you would also specify values for the properties *ConnectionUserName* and *ConnectionPassword*.

The persistence descriptor must be stored in an XML file named *persistence.xml*. This file must be inside a folder named META-INF, and that folder must be in the classpath of the client. Note that the META-INF folder for the demo *JPAStudentInfo* client is in the same folder as the *StudentInfo.class* file. This placement makes sense because the *StudentInfo* client is not part of an explicit package. Since *StudentInfo* can only be executed when its folder is in the classpath (usually, the user's current directory), the META-INF folder will likewise be in the classpath.

9.3.4 The *JPAStudentInfo* Client

A JPA client consists of a domain model and a view. So far we have seen how to write the pieces individually. We now examine what it takes to put it all together.

There are two steps involved in writing the domain model. First, the programmer writes a class for each table in the database, using annotations to specify its mapping to the database. And second, the programmer writes a persistence descriptor that enumerates these classes and gives the necessary connection information. The JPA classes provide everything else.

The view code begins by creating an *EntityManager* object for the persistence unit. The view can then call the entity manager's *find* method to obtain model objects, and call their methods to access the database. The code of Figure 9-13 is a simple template for a JPA view.

The *StudentInfo* code of Figure 9-7 is another example of what a JPA view should look like. That code used our JDBC-based *DatabaseManager* class to build a GUI interface to the database, but the JPA version is almost identical. The one significant difference concerns how transactions begin. In JDBC, a new transaction begins as soon as the previous one completes; JPA, however, does not start a new transaction until the entity manager's *begin* method is called.

The JPA version of the *StudentInfo* client is called *JPAStudentInfo*. This client is part of the SimpleDB distribution, in the folder of demo Derby clients. It is worth comparing the code for these two clients yourself. The view code for the two clients is essentially the same; but as we have seen, the model code for the JPA client is substantially simpler (as well as being more powerful and more robust).

JPA allows us to create domain models simply and effectively, because the built-in EntityManager class does nearly all of the work for us.

9.4 The Java Persistence Query Language

So far we have seen several features of the JPA entity manager. There is one more feature that we have not yet considered, which is the ability of the entity manager to execute queries. This section considers JPA queries in detail.

9.4.1 Adding queries to JPA

Figure 9-16 gives the basic API for the query-related portion of JPA. The entity manager has a method *createQuery*, which takes a query string as argument and returns an object of type *Query*.

EntityManager

```
Query        createQuery(String qry);
```

Query

```
void         setParameter(int position, Object value);
List<Object> getResultList();
Object       getSingleResult();
```

Figure 9-16: Part of the API that supports JPA queries

The *Query* interface is similar to the JDBC interface *PreparedStatement*. Its query string can have parameters; if so, the client calls the *setParameter* method to assign a value to each one. The method *getResultList* executes the query, and returns a list of the entities satisfying the query. If the query returns a single result, then the client instead can call the convenience method *getSingleResult* to retrieve it. In either case, the entity manager adds each returned entity to its persistence context.

Recall that the purpose of JPA is to hide the database from the client, so that the client can operate entirely on Java objects. Consequently, the query mechanism must also hide the database from the client. For example, executing a JPA query returns a list of objects, and not a result set. Moreover, the query must reference objects, and not tables. In other words, the query language cannot be SQL.

A query in JPA is expressed in an SQL-like language called *JPQL*. The primary difference between JPQL and SQL is that JPQL queries are written in terms of model classes instead of database tables. Consider the following simple JPQL query, which retrieves those students who graduated in some unspecified year:

```
select s
from Student s
where s.gradyear = ?1
```

Here, the *from* clause references the class *Student*, and not the table STUDENT. The range variable *s* therefore denotes a *Student* object, and so placing the range variable in the *select* clause tells the query to return a list of *Student* objects. The *where* clause contains the expression *s.gradyear*. This expression refers to the variable *gradyear* of the current *Student* object (and again, not the field *GradYear* of the STUDENT table).

The expression "?1" in the above query denotes a parameter. Recall that a parameter in JDBC is denoted by "?"; the *PreparedStatement* class automatically assigned indexes to each parameter. In JPQL, the programmer assigns indexes to the parameters explicitly.

The entity manager is responsible for translating a JPQL query into SQL. That translation is usually straightforward. To translate the above query to SQL, the entity manager maps the class name *Student* to its table STUDENT, the variable name *gradyear* to its associated field *GradYear*, and the range variable *s* in the *select* clause by the field list *s.**. Recall that the *Student* object returned by the JPQL query will contain the *Dept* object for that student's major. Thus the entity manager will also add a join with the DEPT table to the SQL query it creates.

The JPQL language has many features in common with SQL, such as aggregation, nested queries, and sorting. These features have essentially the same syntax, and so we will not cover them here. Instead, we will focus on three important features of JPQL that are not part of SQL: entity joins, path expressions, and the ability to return multiple kinds of object.

Entity Joins

In SQL, the typical way to join two tables is to equate the foreign key of one table with the key of the other. This technique is not possible in JPQL, because model objects do not contain foreign keys. Instead, a JPQL query will equate objects. Figure 9-17 illustrates the difference between these two join techniques. Each of its two queries retrieve the students who took the basketweaving course: the first query is in SQL, and the other is in JPQL.

```
select s.*
from STUDENT s, ENROLL e, SECTION k, COURSE c
where s.SId = e.StudentId and e.SectionId = k.SectId
and   k.CourseId = c.CId  and c.Title = 'basketweaving'
```

(a) An SQL query

```
select s
from Student s, Enroll e, Section k, Course c
where s = e.student and k = e.section
and   c = k.course  and c.title = 'basketweaving'
```

(b) A JPQL query

Figure 9-17: Queries to retrieve the students who took basketweaving

Consider the term *s = e.student* in the JPQL query. The class *Enroll* has a variable *student* that holds a *Student* object and encodes the relationship between the two classes.

The expression *e.student* therefore denotes a *Student* object, and so the term *s* = *e.student* make sense. In particular, it constrains the two range variables so that *s* refers to the student held by *e*.

Path Expressions

The last line of the JPQL query in Figure 9-17(b) contains the expression:

```
c = k.course and c.title = 'basketweaving'
```

In JPQL, we can combine the two terms in that expression into the single term

```
k.course.title = 'basketweaving'
```

The idea is similar to how method calls are composed in Java. Given the *Section* object *k*, we can "call" its *course* variable to obtain a *Course* object; and from that object, we can "call" its *title* variable to obtain a string. The expression *k.course.title* is called a *path expression*.

Path expressions can often simplify a JPQL query by reducing the need for range variables. Figure 9-18 contains two more queries equivalent to those of Figure 9-17, both of which use path expressions. The query of part (a) uses a path expression in the *where* clause; the query of part (b) shows that you can also use a path expression in the *select* clause.

```
select s
from Student s, Enroll e
where s = e.student and e.section.course.title = 'basketweaving'
```

(a) A path expression in the *where* clause

```
select e.student
from Enroll e
where e.section.course.title = 'basketweaving'
```

(b) A path expression also in the *select* clause

Figure 9-18: Two JPQL queries that use path expressions

Returning Multiple Kinds of Object

The *select* clause of a JPQL query usually contains a single item, which might be either a range variable or a path expression. The reason is that a query is typically used to obtain a collection of objects of some type, as in "retrieve all students who took basketweaving".

Occasionally, however, a client might want to write a query that retrieves tuples of objects. For example, the query of Figure 9-19(a) retrieves pairs of the form [student, section], for each student enrolled in a section offered in 2007.

```
select e.student, e.section
from Enroll e
where e.section.yearOffered = 2007
```

(a) The JPQL query

```
String qry = "select e.student, e.section from Enroll e "
           + "where e.section.yearOffered = 2007";
Query q = em.createQuery(qry);
for (Object[] pair : q.getResultList()) {
   Student s = (Student) pair[0];
   Section k = (Section) pair[1];
   System.out.println(s.getName() + " " + k.getProfessor());
}
```

(b) Extracting the query's result list in Java

Figure 9-19: A JPQL query that retrieves pairs of objects

This query returns a list of objects, where each object is a pair. JPA implements pairs as an array of objects. The code of Figure 9-19(b) illustrates how a client extracts the return objects from the array. The method *getResultList* returns a list of objects. So since the query's select clause mentions a pair of objects, each element of the return list will be of the type *Object[]*. The body of the loop then extracts each object from the array and casts it into the appropriate type.

9.4.2 Using queries in a JPA client

Sometimes when we are told about a new feature, we are not sure about how important the feature is. In particular, JPA queries seem interesting, but are they needed? Can't we get along well enough without them?

There are three reasons why we need queries:

- to obtain an entity without knowing its key value;
- to make the client more readable, by simplifying code that navigates through multiple entities;
- to make the client more efficient, by allowing the entity manager to retrieve multiple kinds of entity in a single database operation.

We shall discuss each situation in detail.

<u>Obtaining an entity without knowing its key</u>

Suppose you want to write a JPA client to print the name of all students that graduated in 2005. How do you obtain the appropriate *Student* entities? If we try to use the *EntityManager* API of Figure 9-12, we encounter a problem. The only way to obtain an entity is by calling the entity manager's *find* method, but in order to do so we need to know the key value. Clearly, that method is not going to be able to help us.

We encountered this issue in Section 9.4.2, where we saw that there is no good way to add the necessary methods to the database manager. JPA's ability to execute queries is the solution we were looking for. Figure 9-20 gives a JPA client that prints the names of the class of 2005. Note how this client resembles a JDBC client, in the way that it executes a query string and loops over the resulting list of records. However, unlike JDBC, the code knows nothing about the database. The query and all of the method calls manipulate objects from the domain model.

```
public class JPAClassOf2005 {
   public static void main(String[] args) {
      EntityManagerFactory emf =
                 Persistence.createEntityManagerFactory("studentdb");
      EntityManager em = emf.createEntityManager();
      EntityTransaction tx = em.getTransaction();
      tx.begin();
      String qry = "select s from Student s where s.gradyear=2005";
      Query q = em.createQuery(qry);
      List classOf2005 = q.getResultList();
      for (Object obj : classOf2005) {
         Student s = (Student) obj;
         System.out.println(s.getName());
      }
      try {
         tx.commit();
      }
      catch (RollbackException e) {
         e.printStackTrace();
         tx.rollback();
      }
      finally {
         em.close();
      }
   }
}
```

Figure 9-20: A JPA client to print the name of students graduating in 2005

Simplifying entity navigation

Suppose now that we want to write a JPA client that prints the name of all students in the same calculus section as student #1. Figure 9-21 shows two different ways to write such a client. (To enhance the comparison, the figure only shows the relevant code fragment in each case.)

```
Student s = em.find(Student.class, 1);
for (Enroll e1 : s.getEnrollments()) {
   Section k = e1.getSection();
   if (k.getCourse().getTitle().equals("calculus"))
      for (Enroll e2 : k.getEnrollments())
         System.out.println(e2.getStudent().getName());
}
```

(a) Using navigation between entities

```
String qry = "select e1.student "
           + "from Enroll e1 "
           + "where e1.section in (select k "
           +                      "from Section k, Enroll e2 "
           +                      "where e2.student.sid = 1 "
           +                      "and k = e2.section "
           +                      "and k.course.title = 'calculus')"

Query q = em.createQuery(qry);
List students = q.getResultList();
for (Object obj : students) {
   Student s = (Student) obj;
   System.out.println(s.getName());
}
```

(b) Using a JPQL query

Figure 9-21: Two ways to find the students in Joe's calculus section

The code fragment of part (a) uses entity navigation. It begins by finding the *Student* object having an ID of 1. It then finds all of the enrollments for that student. For each enrollment, it gets the corresponding section and course, and checks the course title. If the title is "calculus", then it gets the enrollments of that section, and prints the name of each corresponding student.

The code fragment of part (b) uses a nested JPQL query. The inner query returns the section of the calculus course taken by student #1; the outer query returns the names of the students in that section.

The navigational code is somewhat difficult to read; all of the movement between entities makes it hard to keep track of what is happening and why. The advantage of using JPQL is that the relationships between the entities is isolated in that one query. People who are comfortable with SQL tend to find the query-based code easier to grasp.

Improving efficiency

The third reason for using queries is that they can improve the efficiency of the client. Figure 9-21 can be used to illustrate this point as well.

In part (b), the entity manager only needs to make a single call to the database server – it sends the query to the server, and receives the response. On the other hand, consider the navigational code of part (a). Here, the entity manager must contact the server at least three separate times:

- The call to *em.find* will ask the server to return the *Student* data for student #1. The data on that student's enrollments will not be requested, because of lazy evaluation.
- The call to *s.getEnrollments* will ask the server to return the *Enroll* data for that student. The entity manager also will get the *Section* and *Course* (and *Dept*) data corresponding to these enrollments, because of eager evaluation. The entity manager can get this data in a single call to the server.
- The call to *k.getEnrollments* will ask the server to return the *Enroll* data for that section. The entity manager will also ask for the corresponding *Student* data at the same time.

There are two reasons why the query of part (b) will be more efficient. The first reason is that each call to the server is time consuming; multiple calls imply additional communication overhead. The second (and more important) reason is that the single query allows the server to do the most work. Recall the discussion of Section 8.3, where we argued that a JDBC client needs to let the server do as much work as possible. The situation is the same here. The navigational code of part (a) is essentially hard-coding a particular way of evaluating the query. It is almost certainly better to let the server figure out the best way to do it.

In general, a client is more efficient if it uses queries instead of navigation. However, not all navigational code is inefficient. Consider for example the *StudentInfo* client of Figure 9-7. Its *display* method used navigation to find the transcript-related information for a given student. Would it be more efficient to rewrite it to use a query? Suppose that the student has N enrollments. Then it seems like the code goes to the database 2N+1 times: once to get the enrollments, and twice for each enrollment to get its section and course. However, the entity manager will actually retrieve the section and course information at the same time as the enrollments, because those relationships are eagerly evaluated. Thus the entire for-loop will be evaluated in a single call to the database, which is as efficient as you can get. So although it may be better to replace the navigational code by a JPQL query (see Exercise 9.10), efficiency is not one of the reasons.

9.5 Downloading a JPA Implementation

JPA is a client-side library. The job of the entity manager is to give its client programs the illusion that there is a local database of persistent Java objects. When a client performs an operation on one of these objects, the entity manager performs all of the low-level work: It translates the operation into JDBC, communicates with the database server, and (in the case of a query) turns the result set into a list of Java objects that it caches locally.

The database server knows nothing about JPA. The server communicates with all of its clients using JDBC, and a JPA client is no different from any other. The point here is that JPA implementations are database-independent. Any JPA implementation should work with any JDBC-compliant database server[†].

An open-source JPA implementation, called *OpenJPA*, is available at *openjpa.apache.org*. You should download the current version from this site and extract its contents. You will need to add several jar files to your classpath: the file *openjpa-x.y.z.jar* from the top-level folder; and all of the jar files in the *lib* folder.

9.6 Chapter Summary

- Object-oriented programs are difficult to write using JDBC, because a programmer is forced to treat data in terms of rows of values, instead of as a collection of meaningful objects.

- A good object-oriented program should always separate *view classes* from *model classes*. The view classes handle the input and output. The model classes maintain the internal state of the system, and provide methods to change the state and perform computations on it. The classes in the object-relational mapping belong to the model.

- The *impedance mismatch* is the distinction between what a programmer wants and what JDBC supplies. The best way to deal with the impedance mismatch is to write Java classes that encapsulate the JDBC code and provide an object-oriented view of the database. These classes are called the *object-relational mapping* (ORM).

- A good way to implement an object-relational mapping is to use the *Data Access Objects* (DAO) pattern. In this pattern, each class in the ORM has an associated private DAO class, which contains all of the database-specific code.

- The *Java Persistence Architecture* (JPA) is a library that makes it easy to write object-relational mappings. The central component of JPA is the *entity manager*.

- Each JPA client creates its own entity manager. The manager maintains a *persistence context*, which is a cache of the model objects that it has seen. The manager has three important methods:
  - The method *find* retrieves from the database the model object having a specified key value. If the object is currently in the model, then the method can return it directly, without having to access the database.
  - The method *persist* inserts a new record into the database corresponding to the given model object.

[†] Consequently, JPA will not work with SimpleDB. One issue is that SimpleDB does not allow clients to turn off autocommit mode.

  - The method *remove* deletes the record from the database corresponding to the given model object.

- The entity manager performs all changes locally on its persistence context, but for efficiency it may postpone changing the database. When the transaction commits, it batches all changes using as few SQL commands as possible.

- The entity manager code is provided by a JPA vendor. The programmer is responsible for writing the classes that implement the object-relational mapping. The programmer uses annotations to specify the following information about each class:
  - The @Table annotation specifies the database table that the class corresponds to.
  - The @Column annotation specifies the field of that table that a class variable corresponds to.
  - The @Id annotation specifies the key field of the table.
  - The @ManyToOne annotation specifies the foreign key of each relationship.
  - The @OneToMany annotation specifies the inverse relationship, which is needed for bi-directional navigation between objects.

- The *@Table* and *@Column* annotations can be omitted if the name of the class and variables are the same as the name of the table and fields.

- Each model also has a *persistence descriptor*, which is stored in an XML file named *persistence.xml*. The descriptor specifies implementation details such as:
  - the class that implements the entity manager;
  - the driver and connection string for the database; and
  - the model classes associated with the database.

- The entity manager is able to execute JPQL queries. The JPQL language is similar to SQL, except that JPQL queries are written in terms of model classes instead of database tables.

- There are three reasons why a client may want to execute a JPQL query:
  - to obtain an entity without knowing its key value;
  - to make the client more readable, by simplifying code that navigates through multiple entities;
  - to make the client more efficient, by allowing the entity manager to retrieve multiple kinds of entity in a single database operation.

9.7 Suggested Reading

This chapter covers only the basics of the JPA library. Further details can be found in the book [Keith and Schincariol, 2006]. The JPA specification document is also quite readable; it can be found online at *jcp.org/aboutJava/communityprocess/final/jsr220/index.html*. (Be sure to download the persistence document, not the ejbcore document.)

There have been numerous proposals for adding persistent objects into Java, which date back to the very beginnings of the language. The article [Atkinson et al. 1996] is a good representative of early efforts, and raises many of the issues discussed in this chapter. The Java community has also adopted various standard persistence APIs over the years, such as *Entity Beans* and *Java Data Objects*. These standards failed primarily because they were too difficult to use. Instead, nonstandard persistence frameworks such as *Hibernate* (*www.hibernate.org*) have become the choice for large software systems. JPA is an attempt to standardize the best ideas of all these attempts. A more detailed history can be found in [Keith and Schincariol, 2006].

Chapter 9 of [Larman 2005] discusses how to design domain models for an object-oriented system. The book [Alur et al. 2001] describes the DAO and related patterns.

9.8 Exercises

CONCEPTUAL EXERCISES

9.1 Figures 9-8 and 9-9 gives the implementation for the JDBC model classes *Student* and *StudentDAO*. Without looking at the demo client code, design the code for the remaining JDBC model classes.

9.2 Figure 9-14 gives the JPA implementation of *Student*. Without looking at the demo client code, design the code for the remaining JPA model classes.

9.3 Suppose that a JPA client is about to execute a query. Explain why the entity manager may need to write pending updates to the database. Should it write them all? If not, can you give an algorithm to determine which updates to write?

9.4 The *StudentInfo* client reads the entire record set into a JTable. Another option is to connect the JTable to the record set, by creating a *TableModel* object for the result set.
a) Explain why connecting to the record set might be more efficient when the recordset contains a lot of records.
b) What is the main problem with connecting to the record set directly?

PROGRAMMING EXERCISES

9.5 Consider the demo program *FindMajor*.
a) Write the code for the class *DBStudent*, such that the code for Figure 9-1 can be executed as-is.
b) Rewrite the code in Figure 9-1 so that it uses the model classes of Section 9.2.
c) Rewrite the code so that it uses the JPA classes of Section 9.3.

9.6 Revise the model classes of Section 9.2 in the following ways:
a) Include the ability to delete objects.
b) Turn all occurrences of eager evaluation into lazy evaluation.
c) Implement object caching.

d) Add methods *findAllStudents*, *findAllDepts*, etc. to the database manager.
e) Change the code of the DAO find methods so that they use a single SQL query to get all of the data they need from the database.
f) Add the method *rollback* to the class *DatabaseManager*.

9.7 Revise the *JPAStudentInfo* client so that a user is also able to insert new students and delete existing students.

9.8 Consider the bookstore database from Exercise 2.7.
a) Write the object-relational mapping in JPA for this database.
b) Create a the *persistence.xml* file for this mapping.
c) Write a view class that tests out your model.

9.9 Consider the videostore database from Exercise 3.24.
a) Write the object-relational mapping in JPA for this database.
b) Create the persistence.xml file for this mapping.
c) Rewrite your JDBC code from Exercise 8.2 to use JPA.

9.10 Consider the method *display* from the *JPAStudentInfo* demo.
a) Rewrite the method to use a JPQL query instead of object navigation.
b) Do you think that this version of the code is better than the original version? Explain.

10

DATA EXCHANGE

and the javax.sql.rowset *and* javax.xml.transform *packages*

Chapters 8 and 9 focused on the problem of how a user can extract data from a database server. We now consider the problem of how the user can make that data available to non-users of the database. Currently, the most popular approach is to exchange the data in XML format. This chapter examines the process of converting data into (and out of) XML, and the Java tools that simplify the process.

10.1 Sharing Database Data with Non-Users

When we think of database access, we think of a user connecting to a database server and issuing SQL statements. But there are also times when a non-user should be able to interact indirectly with the database server. Consider the following examples:

- *Viewing product information from a website*. Consider the way that the *amazon.com* website lets you retrieve book information. Your browser contacts the web server, which contacts Amazon's database server and sends the formatted results back to you. In other words, the web server is the database user, not you.
- *Submitting information to a website.* Suppose that the university allows prospective freshmen to submit their application via the university's website. The web server takes each submitted application and stores it in a database. Again, the web server is the database user.
- *Moving data between database systems*. Suppose that the database that holds the freshman applications is a private database maintained by the admissions office. As soon as the freshman class has been determined, the office sends the name and major of these new students to the administrator of the university database, who inserts new records into the STUDENT table. In this case, neither the admissions office nor the university DBA need to be users of each other's database.
- *Publicizing data*. Every Spring, the university career center creates a CD containing the name, graduation year, and list of courses taken of each graduating senior. It then sends copies of this CD to job recruiters throughout the country.

In each of these cases, data is being sent from a database user to a non-user (or from a non-user to a user). But what is actually getting sent? Since a non-user does not have a direct connection with the database server, it cannot use a result set to examine the output records one at a time. Instead, the entire output table must be sent to the non-user in some predefined format.

Java uses the term *serialization* to denote the process of encoding an object as a stream of bytes, so that it can be saved in a file or sent over a network. So basically, the topic of this chapter is how to serialize the output of the query.

> *The sender of the data needs to choose a format*
> *in which to serialize that data.*

Consider the last of the above examples. The CD produced by the career center contains, for every graduating senior, the same information as the *StudentInfo* client of Figure 9-6. Figure 10-1 depicts how this data could be formatted as a text file. Each line of the file denotes a course taken by a student. A line contains five fields, denoting the student's name, graduation year, course title, year taken, and grade received. Fields are separated by TAB characters, and records are separated by RETURN characters. To save space, the name and graduation year of each student appears only once, with the first course taken. A record that contains a blank name and graduation year denotes another course taken by that student.

```
ian TAB 2009 TAB calculus     TAB 2006 TAB A  RET

    TAB      TAB shakespeare  TAB 2006 TAB C  RET

    TAB      TAB pop culture  TAB 2007 TAB B+ RET

ben TAB 2009 TAB shakespeare  TAB 2006 TAB A- RET

    TAB      TAB databases    TAB 2007 TAB A  RET

tom TAB 2009 TAB calculus     TAB 2006 TAB C- RET

...
```

Figure 10-1: A possible format for data on graduating students

Although this file seems relatively readable to us, the recipient of the file might have some difficulties understanding it. For example:

- There is nothing in the file to indicate the meaning of the five fields. In particular, the second and fourth fields look like they each denote a year, but it is not clear what the difference between them is.
- Similarly, the meaning of the lines having empty fields is not obvious.
- The file does not explicitly indicate that the TAB and RETURN characters act as delimiters. And how does the file represent a value (such as a course title) that contained a TAB?

In addition to these readability issues, there is also the practical problem of how to process the file. The recipient will need to write a program to read each value in the file and determine the record it belongs to. Such activity is called *parsing* the file. The parsing code could become complex, depending on the technique used to save space, and whether delimiters can appear inside of values.

We can make it easier to understand the file by encoding additional semantics into it. Consider the first example at the beginning of the chapter, in which a non-user requests data from the Amazon web site. The web server not only sends the data, but also *annotates* it with HTML tags. These tags tell the browser how to interpret the data values, and makes it possible for the browser to display the data in a meaningful way.

HTML tags specify how data should be displayed. Such tags are useful for browser-based data access, but not for anything else. We need tags that are data-specific instead of presentation-specific. The language of data-specific tags is called *XML*, and is the subject of the next section.

10.2 Saving Data in an XML Document

The acronym XML stands for *eXtendable Markup Language*. The term "markup language" means that an XML document consists of two things:

- data, and
- *elements* that annotate the data.

The term "extendable" means that XML doesn't have a fixed set of elements; users are free to define their own.

> *An XML document uses elements to annotate data.*
> *Elements are data-specific, not presentation-specific.*

Because XML is extendable, it can be used in any context. For example, the JPA persistence descriptor of Figure 9-15 used XML to specify the configuration information needed by the entity manager. That descriptor used elements named *persistence*, *persistence-unit*, *provider*, *class*, and so on. Note that these names mean nothing special to the computer. Whoever designed the persistence-descriptor format chose those names in the hope that they would be readily understandable to humans.

In this chapter, we will use XML to represent data from the database. Figure 10-2 shows how the graduating-student data from Figure 10-1 could be written as an XML document. That document has two types of element: *Enrollments* and *CourseTaken*. There is one *CourseTaken* element for each course taken by a graduating student. And there is only one *Enrollments* element, which groups together all of the *CourseTaken* elements.

```
<Enrollments>
   <CourseTaken sname="ian"  gradyear="2009"  title="calculus"
                yeartaken="2006"  grade="A"  />
   <CourseTaken sname="ian"  gradyear="2009"  title="shakespeare"
                yeartaken="2006"  grade="C"  />
   <CourseTaken sname="ian"  gradyear="2009"  title="pop culture"
                yeartaken="2007"  grade="B+" />
   <CourseTaken sname="ben"  gradyear="2009"  title="shakespeare"
                yeartaken="2006"  grade="A-" />
   <CourseTaken sname="ben"  gradyear="2009"  title="databases"
                yeartaken="2007"  grade="A"  />
   <CourseTaken sname="tom"  gradyear="2009"  title="calculus"
                yeartaken="2006"  grade="C-" />
   ...
</Enrollments>
```

Figure 10-2: An XML document that represents data on graduating students

Every XML document must contain exactly one outermost element. In Figure 10-2, this element is *Enrollments*. The outermost element is also known as the *root element*. All element names in XML are case sensitive.

XML uses *tags* to specify elements. There are two tagging techniques illustrated in this figure: *bracketed tags* and *empty tags*.

Bracketed tags are a pair of tags that specify an element by surrounding its content. The beginning of the element is indicated by a *start tag*, and the end is indicated by a *close tag*. The element *Enrollments* in Figure 10-2 uses bracketed tags. Note how a start tag opens with the character “<”, whereas the close tag opens with the characters “</”. The contents of a bracketed element consists of everything between its start and close tags. In Figure 10-2, the content for *Enrollments* happens to be just other elements, but in general it could also contain untagged text.

The empty-tag technique represents an element by using a single tag, with no other content. Empty tags end with the characters “/>”. The *CourseTaken* elements in Figure 10-2 use this technique. Note that all of the information about each element is contained within its tag, instead of being outside it.

A *CourseTaken* tag uses *attributes* to indicate its data. The keywords *sname*, *gradyear*, *title*, *yeartaken*, and *grade* are attributes of *CourseTaken*; the value for each attribute appears after the equals sign. Attribute values are always quoted, regardless of their type. Spaces are not allowed on either side of the equals sign.

XML elements are implemented using bracketed tags or empty tags.

- *Bracketed tags surround the contents of the element.*
- *Empty tags use attributes to capture the contents of the element.*

Attributes are often used with empty tags, because that is the only way the tags can indicate data. Attributes can also appear in the start tag of a bracketed pair. Figure 10-3 shows a revision of Figure 10-2 in which the *Enrollments* tag contains the attribute *gradyear*. The idea is that the document only needs to express the graduation year once, because all students are graduating in the same year.

```
<Enrollments  gradyear="2009">
   <CourseTaken sname="ian"  title="calculus"
                yeartaken="2006"  grade="A"  />
   <CourseTaken sname="ian"  title="shakespeare"
                yeartaken="2006"  grade="C"  />
   <CourseTaken sname="ian"  title="pop culture"
                yeartaken="2007"  grade="B+" />
   <CourseTaken sname="ben"  title="shakespeare"
                yeartaken="2006"  grade="A-" />
   <CourseTaken sname="ben"  title="databases"
                yeartaken="2007"  grade="A"  />
   <CourseTaken sname="tom"  title="calculus"
                yeartaken="2006"  grade="C-" />
   ...
</Enrollments>
```

Figure 10-3: Revising Figure 10-2 so that the graduation year appears once

There are many ways to represent the same data in XML. Figures 10-2 and 10-3 represent the data as a sequence of enrollments. This representation is rather “flat”, in the sense that it resembles a relational table. A more nested approach would be to represent the data as a list of students, where each student has a list of courses taken. This approach is shown in Figure 10-4.

```
<GraduatingStudents>
   <Student>
      <SName>ian</SName>
      <GradYear>2009</GradYear>
      <Courses>
         <Course>
            <Title>calculus</Title>
            <YearTaken>2006</YearTaken>
            <Grade>A</Grade>
         </Course>
         <Course>
            <Title>shakespeare</Title>
            <YearTaken>2006</YearTaken>
            <Grade>C</Grade>
         </Course>
         <Course>
            <Title>pop culture</Title>
            <YearTaken>2007</YearTaken>
            <Grade>B+</Grade>
         </Course>
      </Courses>
   </Student>
   <Student>
      <SName>ben</SName>
      <GradYear>2009</GradYear>
      <Courses>
         <Course>
            <Title>shakespeare</Title>
            <YearTaken>2006</YearTaken>
            <Grade>A-</Grade>
         </Course>
         <Course>
            <Title>databases</Title>
            <YearTaken>2007</YearTaken>
            <Grade>A</Grade>
         </Course>
      </Courses>
   </Student>
   <Student>
      <SName>tom</SName>
      <GradYear>2009</GradYear>
      <Courses>
         <Course>
            <Title>calculus</Title>
            <YearTaken>2006</YearTaken>
            <Grade>C-</Grade>
         </Course>
      </Courses>
   </Student>
   ...
</GraduatingStudents>
```

Figure 10-4: Using nested XML tags to represent data on graduating students

Figure 10-4 does not use attributes to represent data values. Each data value is stored as text within a pair of start and close tags, instead of as an attribute value inside of an empty tag. Consequently, these values do not need to be quoted. However, whitespace around data values *is* meaningful. For example, suppose that we modified the document so that Ian's name was represented like this:

```
<SName>  ian  </SName>
```

Then the XML processing tools will interpret his name as having those additional spaces, which is probably not what we want.

The nested-tag representation of Figure 10-4 resembles a tree, in which the leaf nodes are the document's text, and the internal nodes are the elements that annotate the text. Figure 10-5 depicts this tree structure.

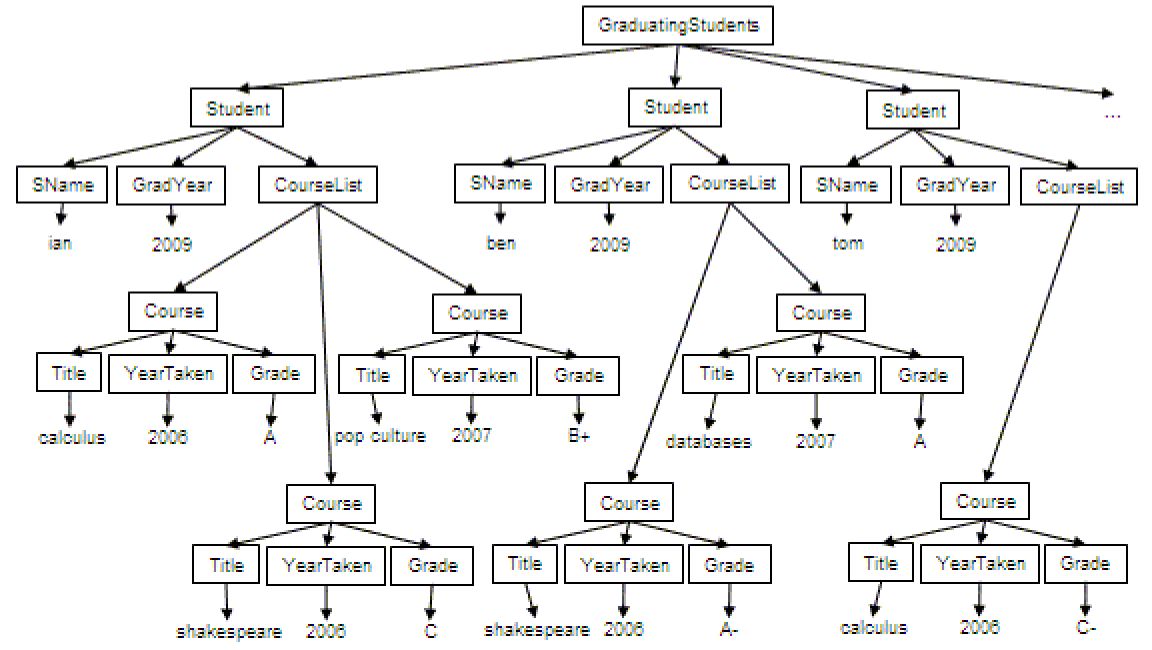


Figure 10-5: A tree representation of the XML document of Figure 10-4

In general, any XML document can be written as a tree. We say that XML supports a *hierarchical* view of the data.

We are now at the point where we can compare the file format of Figure 10-1 with the XML formats of Figures 10-2 to 10-4. In each case, the data is stored as text. The principal difference is that Figure 10-1 indicates field and record boundaries by means of special delimiter characters, whereas XML indicates the boundaries by using tags. The XML approach has three important benefits:

- The tags make it much easier for humans to understand the meaning of the data.
- The tags make it easy for a designer to format the data in whatever hierarchical structure seems most appropriate.
- The numerous classes in the Java XML libraries makes it much easier to create and manipulate a document in XML format.

This last bullet point is especially important. An XML file might be easy for a human to read, but its hierarchical structure can be difficult to parse. The reason why XML is so popular is that there are many Java tools to do all of the hard work for us. The rest of this chapter investigates these tools.

10.3 Restructuring an XML Document

One of the most useful aspects of XML is the ease with which an XML document can be transformed into other documents, such as HTML files, other XML documents, or even text files. The key to this ability is the language *XSLT*, which stands for *eXtendable Stylesheet Language Transformation*.

10.3.1 Basic XSLT

An XSLT program is usually called a *stylesheet*. A stylesheet is nothing but an XML document that uses a certain set of elements. Each of these elements is interpreted as a command to transform some part of the input XML document.

The easiest way to learn XSLT is by example. Figure 10-6(a) gives a simple XSLT program. The program assumes that its input is an XML document having the structure of Figure 10-4. Its output is an XML document that lists the names of the students mentioned in the input document. In particular, if Figure 10-4 is used as the input to the program, then Figure 10-6(b) shows the program's output.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">

   <xsl:output method="xml" indent="yes"/>

   <xsl:template match="GraduatingStudents">
      <StudentNames>
      <xsl:apply-templates select="Student"/>
      </StudentNames>
   </xsl:template>

   <xsl:template match="Student">
      <Name><xsl:value-of select="SName"/></Name>
   </xsl:template>

</xsl:stylesheet>
```

(a) The XSLT program

```
<StudentNames>
   <Name>ian</Name>
   <Name>ben</Name>
   <Name>tom</Name>
</StudentNames>
```

(b) The output of the program

Figure 10-6: Extracting the student names from Figure 10-4

This program uses five XSLT element types: *stylesheet*, *output*, *template*, *apply-templates*, and *value-of*. Each of these elements is prefixed by the namespace designator *xsl*. That is, the tag

```
<xsl:stylesheet ... >
```

denotes the *stylesheet* element in the *xsl* namespace. Namespaces in XML serve the same purpose as packages in Java – they avoid problems with conflicting element names. In an XSLT document, the *xsl* namespace has an additional purpose: It identifies the XSLT commands.

The *stylesheet* element is the root element of an XSLT document. Its start tag has two attributes. The first attribute defines the xsl namespace in terms of a URL. This URL is used only as a means of identification; the XSLT interpreter does not actually visit that site. The second attribute specifies the version of XSLT that this program is written for. These attributes and their values are an example of "boilerplate" code; that is, they can just be copied into every XSLT document you write.

The *output* element tells the XSLT interpreter what kind of output it should create. The value of the *method* attribute is usually either "xml", "html", or "text". The *indent* attribute specifies whether the interpreter should add whitespace to the output in order to make it easy to read.

The *template* elements are the heart of the XSLT program. Each template can be thought of as a method that gets called when the interpreter wants to transform an element. The program contains two templates: one for transforming *GraduatingStudents* elements, and one for transforming *Student* elements.

The XSLT interpreter processes its input by transforming the root element of the document tree. The interpreter looks for a template that matches that element, and performs the transformation indicated by that template, as follows:

- Anything that is not an xsl command is output as is.
- The *apply-templates* command returns the output that results from recursively transforming all child nodes that match the specified element.
- The *value-of* command returns the contents of all matching child nodes.

If the root has more than one child that matches the specified element in the *apply-templates* or *value-of* command, then the output is the catenation of the outputs for those children.

For example, consider what happens when the program of Figure 10-6(a) is applied to the XML document of Figure 10-4. The root element is *GraduatingStudents*, and so the first template is called. That template outputs three things: a start tag for *StudentNames*, the output from transforming the *Student* child elements, and a close tag for *StudentNames*. The second template gets called for each *Student* element. That template also outputs three things: a start tag for *Name*, the contents of the *SName* child, and the close tag for *Name*. The result of this transformation appears in Figure 10-6(b).

This use of templates is essentially a recursive tree traversal. The *template* command transforms an internal node of the tree, and the *apply-templates* command recurses down to the node's children. The *value-of* command transforms the leaves of the tree, and is the base case of the recursion.

This recursive approach is often the best way to write an XSLT program. However, XSLT also has a command *for-each*, which can be used for iterative coding. For example, Figure 10-7 shows how the previous program can be rewritten using *for-each*.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="GraduatingStudents">
      <StudentNames>
      <xsl:for-each select="Student">
            <Name><xsl:value-of select="SName"/></Name>
      </xsl:for-each>
      </StudentNames>
  </xsl:template>

</xsl:stylesheet>
```

Figure 10-7: An iterative XSLT program to extract student names

This code calls the *for-each* command instead of *apply-templates*. The body of that command is executed for each matching child element. Note that the body of the *for-each* command in Figure 10-7 is the same as the body of the *Student* template in Figure 10-6.

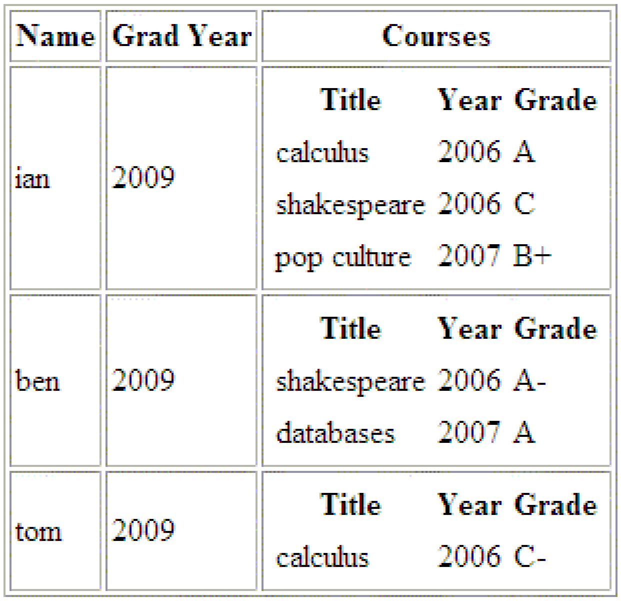
10.3.2 Transforming XML into other formats

An XML document's only job is to hold information – It is completely neutral about what users do with this information. An XSLT program is often used to transform an XML document into something that can be used for a particular purpose. In this section we shall examine two XSLT programs that operate on the XML document of Figure 10-4. One program creates an HTML file for displaying the information in a browser; and the other program creates a file of SQL commands, for inserting the information into a database.

Transforming XML into HTML

An HTML document is structurally very similar to an XML document; the only real difference is that HTML uses its own set of tags. It is thus very easy to write an XSLT program to transform XML into HTML; all the program has to do is replace one kind of tag with another.

Suppose that we wish to display the document of Figure 10-4 in a browser. In particular, we want the HTML document to display a table containing one row for each student, and each row should have a subtable listing the courses taken. Figure 10-8 depicts the desired output.



Name	Grad Year	Courses		
ian	2009	**Title**	**Year**	**Grade**
		calculus	2006	A
		shakespeare	2006	C
		pop culture	2007	B+
ben	2009	**Title**	**Year**	**Grade**
		shakespeare	2006	A-
		databases	2007	A
tom	2009	**Title**	**Year**	**Grade**
		calculus	2006	C-

Figure 10-8: A browser-based view of the student data

Figure 10-9 gives the corresponding XSLT program. This code has a template for each internal node in the document tree.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">

  <xsl:output method="html"/>

  <xsl:template match="GraduatingStudents">
   <html><body>
   <table border="1" cellpadding="2">
   <tr> <th>Name</th> <th>Grad Year</th> <th>Courses</th> </tr>
    <xsl:apply-templates select="Student"/>
    </table>
    </body></html>
  </xsl:template>

  <xsl:template match="Student">
   <tr> <td> <xsl:value-of select="SName"/> </td>
        <td> <xsl:value-of select="GradYear"/></td>
        <td> <xsl:apply-templates select="Courses"/> </td>
   </tr>
  </xsl:template>

  <xsl:template match="Courses">
   <table border="0" cellpadding="2" width="100%">
   <tr> <th width="50%">Title</th> <th>Year</th> <th>Grade</th> </tr>
    <xsl:apply-templates select="Course"/>
    </table>
  </xsl:template>

  <xsl:template match="Course">
   <tr> <td> <xsl:value-of select="Title"/> </td>
        <td> <xsl:value-of select="YearTaken"/>  </td>
        <td> <xsl:value-of select="Grade"/> </td>
   </tr>
  </xsl:template>

</xsl:stylesheet>
```

Figure 10-9: An XSLT program to transform Figure 10-4 into HTML

The *GraduatingStudents* template generates HTML code for the outer table and its header row. Its *apply-templates* command calls the *Student* template once for each of the *Student* children.

The *Student* template generates the HTML code for one row of the table. It uses the values of *SName* and *GradYear* to fill in the first two columns, and it calls the *Courses* template to create the inner table.

The *Courses* template is similar to *GraduatingStudents*. It creates the inner table and header row, and its *apply-templates* command calls the *Course* template to create each data row.

The *Course* template is similar to *Student*. It uses the values of *Title*, *YearTaken*, and *Grade* to fill in the columns of the current row of the inner table.

Creating a File of SQL Commands

Suppose that the recipient of an XML document wants to insert the data into a database. A reasonable approach is to transform the XML document into a file of SQL *insert* commands. The recipient can then write a JDBC program to read the file one line at a time, and execute each command as it is read in (as in Exercise 8.6).

For example, suppose that a company receives a CD from the university containing the document of Figure 10-4. The company has a 2-table database containing applicant data, having the following schema:

```
APPLICANT(AppName, GradYear)
COURSE_TAKEN(AppName, CourseTitle, Grade)
```

Figure 10-10 shows the kind of SQL commands that the company would like to generate. Figure 10-11 gives the XSLT code to generate these commands.

```
insert into APPLICANT (AppName, GradYear) values('ian', 2009 );
insert into APPLICANT (AppName, GradYear) values('ben', 2009 );
insert into APPLICANT (AppName, GradYear) values('tom', 2009 );

insert into COURSE_TAKEN(AppName, Title, Grade)
       values('ian', 'calculus', 'A');
insert into COURSE_TAKEN(AppName, Title, Grade)
       values('ian', 'shakespeare', 'C');
insert into COURSE_TAKEN(AppName, Title, Grade)
       values('ian', 'pop culture', 'B+');
insert into COURSE_TAKEN(AppName, Title, Grade)
       values('ben', 'shakespeare', 'A-');
...
```

Figure 10-10: SQL insert commands corresponding to Figure 10-4

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                  version="2.0">

   <xsl:output method="text"/>

   <xsl:template match="GraduatingStudents">
      <xsl:apply-templates select="Student"/>
      <xsl:apply-templates select="Student/Courses/Course"/>
   </xsl:template>

   <xsl:template match="Student">
      insert into APPLICANT (AppName, GradYear) values(
                            '<xsl:value-of select="SName"/>',
                            <xsl:value-of select="GradYear"/>);
   </xsl:template>

   <xsl:template match="Course">
      insert into COURSE_TAKEN(AppName, Title, Grade) values(
                              '<xsl:value-of select="../../SName"/>',
                              '<xsl:value-of select="Title"/>',
                              '<xsl:value-of select="Grade"/>');
   </xsl:template>

</xsl:stylesheet>
```

Figure 10-11: An XSLT program to transform Figure 10-4 into SQL insert commands

Note that the code needs to generate one insertion into APPLICANT for each *Student* element, and one insertion into COURSE_TAKEN for each *Course* element. Consequently, the best way to write the code is to have the *Student* and *Course* templates be responsible for generating those statements. The *GraduatingStudents* template will call the *Student* template once for each child, and will call the *Course* template once for each great-grandchild.

The *Student* template uses the values of the student's *SName* and *GradYear* children as the values of its insertion statement. The *Course* template uses the values of the course's *Title* and *Grade* children, as well as the value of its grandparent's *SName* child. Note the single quotes surrounding the *value-of* commands for *SName*, *Title*, and *Grade*. These quotes need to be part of the string constants in the SQL insertion statement, and so they must be explicitly generated by the program.

This code introduces a new feature of XSLT called *XPath expressions*.

An XPath expression identifies the paths in the element tree that satisfy a given condition.

An *apply-templates* or *value-of* command can use an XPath expression to select elements at any desired place in the tree. Consider for example the two *apply-templates* commands in the *GraduatingStudents* template. The first one selects on "Student". As we mentioned, this expression matches all *Student* elements that are a child of the current element. The second command selects on "Student/Courses/Course". This expression denotes a path down the tree, starting from the current element. It matches all *Course* elements such that there is path to it starting from the current element and passing through *Student* and *Courses*. In other words, the expression matches each *Course* element that is a child of a *Courses* element that is a child of a *Student* element that is a child of the current element. In the case of Figure 10-4, this expression winds up matching every *Course* element in the tree.

XPath expressions can also refer to elements higher up in the tree. For example, consider the first *value-of* command in the *Course* template. That command selects on the expression "../../SName". The ".." denotes the parent of the current element. Thus this expression matches the *SName* element(s) that is the child of the parent of the parent of the current element – in other words, the name of the student who took the course.

The language of XPath expressions is rich, and contains many additional features. For example, the language allows you to use wildcard characters when selecting element names; it also lets you select elements based on their content. Details can be found in the references listed at the end of the chapter.

10.3.3 XSLT for nesting and unnesting

XLST is also useful for restructuring data. In this section we illustrate the capabilities of XSLT by writing programs to transform between nested and unnested documents. The program of Figure 10-12 shows how to take the nested structure of Figure 10-4 and flatten it to the structure of Figure 10-2.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">

   <xsl:output method="xml" indent="yes"/>

   <xsl:template match="GraduatingStudents">
      <Enrollments>
         <xsl:apply-templates select="Student/Courses/Course"/>
      </Enrollments>
   </xsl:template>

   <xsl:template match="Course">
      <CourseTaken
         sname="{../../SName}"
         gradyear="{../../GradYear}"
         title="{Title}"
         yeartaken="{YearTaken}"
         grade="{Grade}"
      />
   </xsl:template>

</xsl:stylesheet>
```

Figure 10-12: An XSLT program to transform Figure 10-4 into Figure 10-2

The *GraduatingStudents* template generates the bracketed *Enrollments* tags, and calls the *Course* template for each course taken by each student. The *Course* template simply generates the *CourseTaken* tag and its attribute values.

This program demonstrates how to generate attribute values. Note the use of curly braces. The expression *{Grade}* returns the value of the *Grade* element, and is equivalent to *<xsl:value-of select="Grade"/>*. For technical reasons, however, we cannot use that xsl command to generate an attribute value; thus the curly-brace notation is necessary[†].

Figure 10-13 shows an XSLT program to perform the reverse transformation.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="Enrollments">
    <GraduatingStudents>
    <xsl:for-each-group select="CourseTaken" group-by="@sname">
      <Student>
         <SName><xsl:value-of select="@sname"/></SName>
         <GradYear><xsl:value-of select="@gradyear"/></GradYear>
         <Courses>
            <xsl:for-each select="current-group()">
               <Course>
                  <Title><xsl:value-of select="@title"/></Title>
                  <YearTaken><xsl:value-of
                                   select="@yeartaken"/></YearTaken>
                  <Grade><xsl:value-of select="@grade"/></Grade>
               </Course>
            </xsl:for-each>
         </Courses>
      </Student>
    </xsl:for-each-group>
    </GraduatingStudents>
  </xsl:template>

</xsl:stylesheet>
```

Figure 10-13: An XSLT program to transform Figure 10-2 into Figure 10-4

This code illustrates how an XLST program references the attributes of an element.

[†] The problem stems from the fact that XML does not allow the "<" and ">" characters in attribute values. Although the generated XML file will not contain these characters, the XSLT file would. And since XSLT documents must also be legal XML, those characters are not allowed.

An XSLT program references an attribute by prefixing its name with the "@" character.

For example, the command

```
<xsl:value-of select="@grade"/>
```

returns the value of the *grade* attribute of the current element.

This code also illustrates how the XSLT command *for-each-group* performs grouping. Consider the following line from the program:

```
    <xsl:for-each-group select="CourseTaken" group-by="@sname">
```

The *select* attribute says that the groups should consist of the *CourseTaken* children of the current element. The *group-by* attribute says that these elements should be grouped by the value of their *sname* attribute. The command will then generate its output by executing its body for each group.

In Figure 10-13, the body of the *for-each-group* command creates a *Student* element having the three children *SName*, *GradYear*, and *Courses*. The content of *SName* and *GradYear* are simply taken from the first element of the group (since the values are the same for all elements of the group). The content of the *Courses* element, however, is more complex.

The program needs to generate a *Course* element for each element in the current group – which in this case means each course taken by the particular student. To get these elements, you need to call the built-in function *current-group*. The line of code

```
<xsl:for-each select="current-group()">
```

therefore executes its body for each element in the group. Since the body in this case simply creates a *Course* element, the *for-each* command produces a *Course* element for each course taken by the current student, which is exactly what we want.

10.3.4 Executing XSLT in Java

It is good to be able to write XSLT programs, but we also need to be able to execute them. The Java classes for processing XSLT programs are in the packages *javax.xml.transform* and *javax.xml.transform.stream*.

The *javax.xml.transform* package defines the two interfaces *Source* and *Result*. A *Source* object knows how to read from an XML document, and a *Result* object knows how to handle the output of an XSLT program. Conceptually, a *Source* is similar to a *Reader*, and a *Result* is similar to a *Writer*. The package also defines the abstract class *Transformer*. A *Transformer* object embodies an XSLT program. Its method *transform* executes the program, reading from a *Source* object and writing its output to a *Result* object.

Although there are several ways to use a transformer, the most common way is when the XML document and XSLT program are stored in files. In this case, we can use the classes *StreamSource* and *StreamResult* from the package *javax.xml.transform.stream*. Figure 10-14 gives a Java program that executes an XSLT program stored as a file.

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import java.io.*;

public class TransformXmlFile {
   // Fill in the appropriate file names
   public static String XSLFILE = "...";
   public static String INFILE  = "...";
   public static String OUTFILE = "...";

    public static void main(String[] args) {
      try {
         // Enable the Saxon XSLT v2.0 interpreter
         System.setProperty("javax.xml.transform.TransformerFactory",
                            "net.sf.saxon.TransformerFactoryImpl");

         // Create Source and Result objects for the files
         Source xslcode = new StreamSource(new FileReader(XSLFILE));
         Source input   = new StreamSource(new FileReader(INFILE));
         Result output  = new StreamResult(new FileWriter(OUTFILE));

         // Transform the INFILE to the OUTFILE
         TransformerFactory tf = TransformerFactory.newInstance();
         Transformer trans = tf.newTransformer(xslcode);
         trans.transform(input, output);
      }
      catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Figure 10-14: A Java program to execute an XSLT program

The code involves three straightforward steps:

- First, create the *Source* and *Result* objects. The class *StreamSource* turns a *Reader* into a *Source*, and *StreamResult* turns a *Writer* into a *Result*.
- Second, create the *Transformer* object. To do this, you create a *TransformerFactory* object, and then pass the XSLT program into its *newTransformer* method.
- Finally, call the transformer's *transform* method.

The *TransformerFactory* class doesn't actually contain the code to create a transformer; instead, its methods call the hidden class *TransformerFactoryImpl* to do all of the work. The advantage to this setup is that it is possible to plug in different XSLT interpreters. The system property *java.xml.transform.TransformerFactory* specifies the version of the

TransformerFactoryImpl class that will get called. If that property is not set, then a system default will be used.

The default transformer implementation that ships with the Java 6 distribution contains an outdated XSLT interpreter that supports only the XSLT version 1.0 specification. Most of the examples in this chapter use only version 1.0 features, and so they will work with the default implementation. The one exception is the use of grouping in Figure 10-13, which is new to XLST version 2.0. Consequently, the Java code of Figure 10-14 specifies a different XSLT implementation, namely the Saxon interpreter. To run the Java code, you need to download the Saxon implementation from *www.saxonica.com/products.html*, and add the file *Saxon8.jar* to your classpath.

10.4 Generating XML Data from the Database

The theme of this chapter is how to use XML to send data from a database to a non-user. So far, we have seen that the recipient of an XML document can transform the document in several ways. We now turn to the question of how the sender can create the document.

10.4.1 Generating XML manually

It is actually not very difficult to write a JDBC program to read data from a database and write it to an XML file. Consider Figure 10-15.

```
public class StudentsToXML {
   public static final String OUTFILE = "..."; //fill in this filename

   public static void main(String[] args) {
      Connection conn = null;
      try {
         // Step 1: connect to database server
         Driver d = new ClientDriver();
         String url = "jdbc:derby://localhost/studentdb;create=false;";
         conn = d.connect(url, null);

         // Step 2: execute the query
         Statement stmt = conn.createStatement();
         String qry = "select s.SName, s.GradYear, c.Title, "
                  +          "k.YearOffered, e.Grade "
                  + "from STUDENT s, ENROLL e, SECTION k, COURSE c "
                  + "where s.SId=e.StudentId and e.SectionId=k.SectId "
                  + "and   k.CourseId=c.CId "
                  + "and   s.GradYear = extract(year, current_date)";
         ResultSet rs = stmt.executeQuery(qry);

         // Step 3: write the resultset data as XML
         Writer w = new FileWriter(OUTFILE);
         writeXML(rs, "Student", w);
         rs.close();
      }
      catch(Exception e) {
         e.printStackTrace();
      }
      finally {
         // Step 4: close the connection
         try {
            if (conn != null)
               conn.close();
         }
         catch (SQLException e) {
            e.printStackTrace();
         }
      }
   }

   public static void writeXML(ResultSet rs, String elementname,
                               Writer w) throws Exception {
      w.write("<" + elementname + "s>\n");

      ResultSetMetaData md = rs.getMetaData();
      int colcount = md.getColumnCount();
      while(rs.next()) {
         w.write("\t<" + elementname + ">\n");
         for (int i=1; i<=colcount; i++) {
            String col = md.getColumnName(i);
            String val = rs.getString(col);
            w.write("\t\t<" + col + ">" + val + "</" + col + ">\n");
         }
         w.write("\t</" + elementname + ">\n");
      }
      w.write("</" + elementname + "s>\n");
   }
}
```

Figure 10-15: A Java program to create an XML document from the database

Step 2 of this code opens a result set that returns a record for each course taken by a graduating student. Step 3 then passes this result set to the local method *writeXML*, which constructs the XML tags and writes them to the writer. Figure 10-16 shows what the output of this program would look like.

```
<Students>
   <Student>
      <SNAME>ian</SNAME>
      <GRADYEAR>2009</GRADYEAR>
      <TITLE>calculus</TITLE>
      <YEAROFFERED>2006</YEAROFFERED>
      <GRADE>A</GRADE>
   </Student>
   <Student>
      <SNAME>ian</SNAME>
      <GRADYEAR>2009</GRADYEAR>
      <TITLE>shakespeare</TITLE>
      <YEAROFFERED>2006</YEAROFFERED>
      <GRADE>C</GRADE>
   </Student>
   ...
</Students>
```

Figure 10-16: Some of the output of Figure 10-15

The method *writeXML* is clearly the interesting portion of the program. Note that the method is independent of the query that produced the result set, because it uses the result set's metadata to determine the number of fields and their names. The name of the element for each record (here, *Student*) is the only element name that must be passed into the method. We use the convention that the name of the top-level element is the plural of that name.

10.4.2 Generating XML using *WebRowSet*

Although the method *writeXML* in Figure 10-15 is relatively easy to write, we can actually do without it. The Java package *javax.sql.rowset* contains an interface named *WebRowSet*, which also has a method *writeXML* that does essentially the same thing.

It is trivial to rewrite the above code to use *WebRowSet*. All you need to do is remove the code that calls the local *writeXML* method:

```
            writeXML(rs, "Student", w);
```

and replace it with the following code, which creates *WebRowSet* object, populates it with the contents of the result set, and calls its *writeXML* method:

```
            WebRowSet wrs = new WebRowSetImpl();
            wrs.populate(rs);
            wrs.writeXML(w);
```

The *WebRowSetImpl* class generates XML documents having a very different structure from what we have seen so far. Figure 10-17 shows some of the document produced by *WebRowSetImpl*.

```
<webRowSet xmlns="http://java.sun.com/xml/ns/jdbc"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation=
                       "http://java.sun.com/xml/ns/jdbc/webrowset.xsd">

  <properties>
    ...
    <isolation-level>2</isolation-level>
    <read-only>true</read-only>
    <rowset-type>ResultSet.TYPE_SCROLL_INSENSITIVE</rowset-type>
    ...
  </properties>

  <metadata>
    <column-count>5</column-count>
    <column-definition>
      <column-index>1</column-index>
      <auto-increment>false</auto-increment>
      <case-sensitive>false</case-sensitive>
      <currency>false</currency>
      <nullable>1</nullable>
      <column-display-size>10</column-display-size>
      <column-name>SNAME</column-name>
      <column-precision>10</column-precision>
      <column-scale>0</column-scale>
      <table-name>STUDENT</table-name>
      <column-type>12</column-type>
      <column-type-name>VARCHAR</column-type-name>
      ...
    </column-definition>
    ...
  </metadata>

  <data>
    <currentRow>
      <columnValue>ian</columnValue>
      <columnValue>2009</columnValue>
      <columnValue>calculus</columnValue>
      <columnValue>2006</columnValue>
      <columnValue>A</columnValue>
    </currentRow>
    <currentRow>
      <columnValue>ian</columnValue>
      <columnValue>2009</columnValue>
      <columnValue>shakespeare</columnValue>
      <columnValue>2006</columnValue>
      <columnValue>C</columnValue>
    </currentRow>
   ...
 </data>
</webRowSet>
```

Figure 10-17: A portion of the XML document produced by *WebRowSetImpl*

This document has the following elements:

- The root element of the document is *webRowSet*. This element has three child elements: *properties*, *metadata*, and *data*.
- The *properties* element describes general synchronization-related properties about the underlying result set.
- The *metadata* element describes each column of the output table; this metadata is similar to what can be obtained from the *ResultSetMetaData* interface in JDBC.
- The *data* element describes the values of each row of the output table. The tags for the rows and fields are general-purpose: Each row is annotated with a *CurrentRow* tag, and each field is annotated with a *columnValue* tag.

Unlike the analogous class *TransformFactoryImpl*, the class *WebRowSetImpl* is not part of the Java 6 distribution; instead, it is an optional class that needs to be downloaded separately. In particular, you need to download the file *rowset.jar* from the site *java.sun.com/products/jdbc/download.html*, and add it to your classpath. The class *WebRowSetImpl* is in the package *com.sun.rowset*, which needs to be imported in any Java code that uses it.

10.4.3 Making use of rowsets

Although the XML document produced by *WebRowSet* is difficult for humans to read, it turns out to be excellent for data transfer. In this section we shall see why.

The interface *WebRowSet* is a subinterface of *RowSet*, which is a subinterface of *ResultSet*. Consequently, if you have a *RowSet* (or *WebRowSet*) object, you can use it in a JDBC program as if it were a result set. The difference between a result set and a rowset has to do with the connection to the database server. A result set requires an active connection to the server, so that its methods can contact the server when needed. A rowset, on the other hand, obtains all of its data from the server when it is created; consequently, methods on a rowset never need to contact the server. In other words, a *RowSet* object holds the entire output of a query.

In addition to the method *writeXML*, the *WebRowSet* interface also has the method *readXML*. This method performs the opposite action: It reads the XML document produced by *writeXML*, constructs a collection of records from it, and then populates the rowset with them. The elements in this rowset can be processed the same as for any result set. The code fragment of Figure 10-18 illustrates the method in use.

```
WebRowSet wrs = new WebRowSetImpl();
wrs.readXML(new FileReader(INFILE));
while (wrs.next()) {
   // process the current record
}
wrs.close();
```

Figure 10-18: Reconstructing the rowset from the serialized XML file

Using Java terminology, we would say that the document produced by the *writeXML* method serializes the rowset as a text file, and the *readXML* method deserializes it. This ability explains why the XML produced by *writeXML* is so extensive – the *readXML* method needs all of the elements in that XML document to fully reconstruct the *RowSet* object.

The XML document produced by *WebRowSet* is a versatile solution to the problem of sending data. If the recipient knows JDBC, she can write a Java program to extract the data into a rowset and process the rowset as desired, as in Figure 10-18. Otherwise, she can write an XSLT program to transform the document into a more desirable format.

10.5 Chapter Summary

- In order for a database user to send data to a non-user, the sender must choose a format to send the data in. The best way to format the data is in XML.

- An XML document consists of *elements* that annotate the data. Elements can be nested, which means that an XML document can be written as a tree. We say that XML supports a *hierarchical* view of the data.

- XML uses *tags* to specify elements. XML elements are implemented using *bracketed tags* or *empty tags*. Bracketed tags surround the content of the element. Empty tags capture the content as attribute values.

- The XML approach has three important benefits:
    - The tags make it easy for humans to understand the meaning of the data.
    - The tags make it easy for a designer to format the data in whatever hierarchical structure seems most appropriate.
    - Java contains built-in classes that make it easy to create and manipulate XML documents.

- The *XSLT language* specifies how an XML document can be transformed into other documents, such as HTML files, other XML documents, or text files. An *XSLT program* is an XML document in which certain elements are interpreted as commands.

- The most important XSLT commands are:
    - *stylesheet*, which is always the root element of an XSLT program.
    - *output*, which specifies whether the output is in xml, html, or text.
    - *template*, which specifies how to transform a specific element.
    - *apply-templates,* which recursively transforms elements related to the current element.
    - *value-of,* which returns the contents of the matching elements.
    - *for-each,* which executes its body for each element in the specified list.
    - *for-each-group,* which executes its body for each specified group.

- An *XPath expression* identifies paths in the element tree satisfying a given condition. An XSLT command uses an XPath expression to select the next nodes to be processed.

- An XSLT program can be executed in Java by creating a *Transform* object for it and calling that object's *transform* method.

- There are two reasonable ways to extract data from a result set into an XML document:
    - Write JDBC code to iterate through the result set, writing XML tags for each output record;
    - Use the result set to populate a *WebRowSet* object, and then ask that object to serialize itself as XML.

10.6 Suggested Reading

This chapter examined ways to use XML documents to exchange database information, and some Java tools that support this exchange. In doing so, we have left out a large amount of material related to XML, Java, and database systems.

One of the most difficult aspects of using XML for data exchange is making it easy for the recipient to use the XML document. This chapter covered the use of XSLT to transform the document. Another, more general approach is for the recipient to traverse the document using an XML parser; the Java API known as *JAXP* (in the package *javax.xml.parsers*) provides a uniform way to access different parsers. Another approach is to bind the XML document to a tree of Java objects, and then traverse these Java objects; this API is called *JAXB*, and is in the package *javax.xml.bind*. The book [McLaughlin 2006] covers these and other APIs in detail.

The XSLT language plays a central role in data exchange. The language has many more features than we are able to cover in this chapter. The books [van Otegem 2002] and [Mangano 2005] provide comprehensive coverage of XSLT and XPath.

Finally, we have not touched on the possibility of storing XML data directly in a database. Many commercial relational database systems support a data type that allows XML documents to be treated as SQL values. There are also database systems that are designed specifically to store XML data. The articles [McHugh et al. 1997] and [Vakali 2005] address these issues.

10.7 Exercises

CONCEPTUAL EXERCISES

10.1 Draw trees corresponding to the XML documents of Figures 9-15, 10-2, 10-16, and 10-17.

10.2 Recall that an XSLT program is an XML document. Write the programs of Figures 10-6(a), 10-9, and 10-13 as trees.

10.3 Explain why *ResultSet* objects cannot be serialized.

10.4 One way to serialize a result set is to insert all of its records into a collection, and then serialize the collection. How does this strategy compare with serialization using a rowset?

10.5 Many database systems know how to load data from an Excel spreadsheet. So yet another way to serialize a result set is to save the output table as an Excel spreadsheet.
a) Is this a good idea? Analyze the tradeoffs.
b) Many scientific data sets are available as Excel files. Would you recommend that these files be reformatted as XML? Suggest a good way to transform an Excel file into an XML file. What issues need to be addressed?

10.6 Suppose you are starting your own e-commerce business, and you want to accept orders by email. However, you want to automate order processing as much as possible. For example, you certainly don't want to have to retype each email order into your ordering database. Explain how you would use XML to help with the automation.

PROGRAMMING EXERCISES

10.7 Consider the third example of Section 10.1, where the admissions office sends data on each new student to the university DBA.
a) Assume that the admission office's database contains a table APPLICANT, having the following schema:

```
APPLICANT(SName, Major, Rating, ...)
```

The value of *Major* is a string that contains the department name. Write a Java program to create an XML file that contains the value of *SName* and *Major* for those students having a rating of 5.
b) Suppose that the university DBA has received the file from part (a). Write a Java program that uses this file to insert appropriate records into STUDENT. The program will need to assign an ID and graduation year to each student.

10.8 The university wants to create a web page that lists the data about the courses offered this year. The web page should be organized by course; for each course, it should give the title of the course, as well as a list of information about each section being offered.
a) Write a Java program that generates an appropriate XML file for this data.
b) Write an XSLT program that transforms this XML file into an appropriate HTML file.

10.9 Figure 10-11 gives an XSLT program that generates SQL insert statements, assuming that the input has the nested structure of Figure 10-4. Rewrite the program under the assumption that the input has the flat structure of Figure 10-2.

10.10 Figure 10-4 organizes the enrollment data by student. Another way to organize it is by course. The XML document could have one element for each course, with each *Course* element containing a list of students who took that course.
a) Draw the XML file equivalent to Figure 10-4 that has this structure.
b) Write an XSLT program to generate this file from the file of Figure 10-2.
c) Write an XSLT program to generate this file from the file of Figure 10-4. (Warning: This is much harder than (b).)
d) Write a JDBC program to output the file by querying the university database.

10.11 Consider restructuring Figure 10-4 to use attributes. In particular, the element *Student* should have the attributes *sname* and *gradyear*, instead of the child elements *SName* and *GradYear*.
a) Rewrite Figure 10-4 to reflect this changed structure.

b) Explain why the element *Courses* is no longer needed.
c) Rewrite the XSLT code of Figures 10-12 and 10-13 to reflect these changes.

10.12 Write an XSLT program to translate the XML of Figure 10-17 to that of Figure 10-16.

10.13 The XML file written by *WebRowSet* contains a lot of elements. Many of these elements are actually optional, and are not needed by the *readXML* method.
a) Determine by trial and error which elements are necessary. That is, write a Java program that calls *readXML* on a file written by *WebRowSet*. Then manually remove elements from the file until the program throws an exception.
b) You should discover that a lot of the metadata elements are not necessary. What happens when your Java program calls *getMetaData* to get the *ResultSetMetaData* object, and then tries to execute its methods?
c) Write an XSLT program to transform the XML of Figure 10-16 to that of Figure 10-17. You do not need to generate non-necessary elements.

10.14 A class *WebRowSet* implements the *Serializable* interface. That is, if we have a populated *WebRowSet* object, we can call the method *writeObject* (from class *ObjectOutputStream*) to write the object to a file; and given the file, we can call the method *readObject* (from class *ObjectInputStream*) to recreate the rowset.
a) Write a Java program similar to Figure 10-15, which creates a rowset of graduating students and serializes it using *writeObject*. Then write a program that reads the resulting output file using *readObject* and prints its contents.
b) What is the difference between the file generated from *writeObject* and the file generated from *writeXML*?
c) Explain why this strategy is not as good as using the *writeXML* and *readXML* methods to do the serialization.

11
WEBSERVER-BASED DATABASE CLIENTS
and the package javax.servlet

All of the database clients that we have examined so far are Java programs that users execute directly. We call these programs *stand-alone applications*. In this chapter we look at the more common situation where users access the database indirectly, via a web browser; we call these programs *webserver-based applications*. On one hand, applications that make use of a web browser have to deal with a very different set of issues: the Java code must be able to run on a web server, and must communicate with the user via HTML code. But on the other hand, their interaction with the database server is for the most part unchanged. This chapter examines these issues.

11.1 Types of Database Clients

In a client-server application, the database client and database server each have their own responsibilities. The database server is responsible for accessing the database. It schedules client requests, determines the most efficient way to execute them, and ensures that the database does not get corrupted. The client is responsible for contacting the server. It determines what data is needed to perform its task and what requests are needed to obtain that data. It must know what database server(s) it needs to connect to, and must decide when to open and close the connections.

There are several ways to build a database client. One way is as a *stand-alone application*. A stand-alone application is code that lives on the client machine and accesses the database server directly. All of the clients that we have seen so far are of this type. Its architecture is depicted in Figure 11-1.

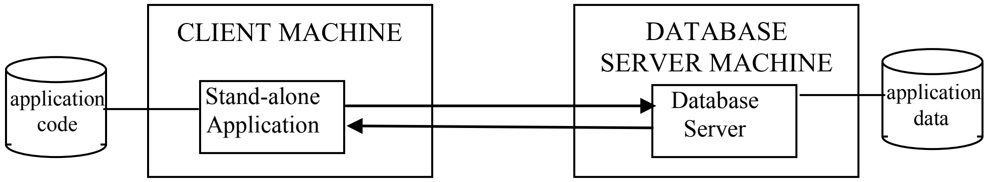


Figure 11-1: The architecture of a stand-alone database client

A second way to build a database client is to incorporate a web browser and web server. In this case, the application is structured as a sequence of *requests* from the browser to the web server. The web server may process a request by returning the contents of a file (a *static* request), or by executing a program and returning its output (a *dynamic* request). Programs that are executed by a dynamic request are able to interact with the database server.

Figure 11-2 depicts the architecture. This figure shows the web server and database server on different machines, but of course they could be (and often are) on the same machine.

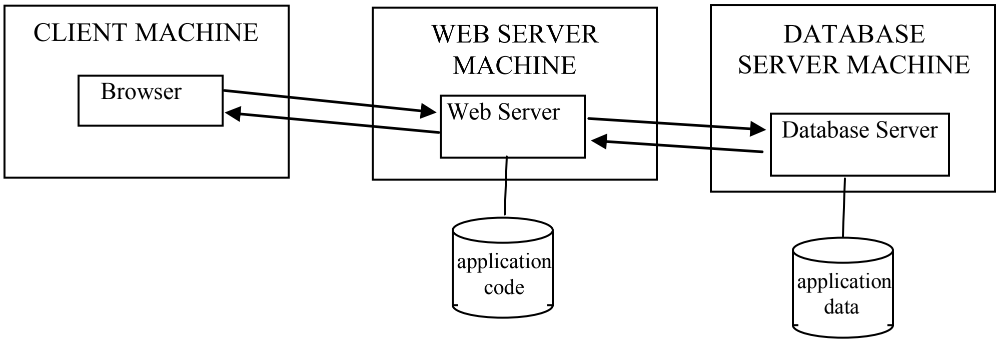


Figure 11-2: The architecture of a webserver-based database client

In a webserver-based client, the browser knows nothing about the database or the application. All calls to the database occur from the code on the web server, and the database output gets assimilated into the response sent back to the browser. The browser has no idea if the response it receives comes from a static request, or from the output of a program that queries the database. For example, consider any e-commerce web site, such as *amazon.com*. If you view a page for a particular item, there is no indication that it was constructed from a dynamic request that accessed a database; the page source consists entirely of HTML code that tells the browser what to display and how to display it.

The webserver-based approach requires the interaction of three programs: the browser, the web server, and the database server. Of these, the browser is a client of the web server, and the web server is a client of the database server. Note that the web server acts as both a server (to the browser) and a client (of the database).

In a webserver-based application, the browser is the client of the web server, and the web server is the client of the database server.

These two approaches to building a database client are at opposite ends of a spectrum. The stand-alone approach creates what is called a *fat client*, because all of the application functionality lives on the client machine. On the other hand, the webserver-based approach creates a *thin client*, because none of the application functionality lives on the client.

11.2 Interacting with a Web Server

11.2.1 Static requests

Suppose that you type the following URL into your browser:

```
http://java.sun.com:80/docs/books/tutorial/jdbc/index.html
```

The browser interprets this URL by splitting it into four pieces, as follows.

`http:` This piece tells the browser to contact the server using the HTTP protocol, which is the protocol used by web servers. Browsers are also able to talk to other kinds of servers. For example, a URL beginning "ftp:" would instruct the browser to talk to the server using the FTP protocol.

`//java.sun.com`
This piece specifies the machine where the server resides.

`:80` This piece specifies the network port that the server is listening on. By convention, web servers listen on port 80, but in fact any port is possible.

`/docs/books/tutorial/jdbc/index.html`
This piece specifies the location of the desired document on the web server.

The browser processes this URL by sending an *HTTP request* to the server on the machine *java.sun.com* that is listening at port 80. An HTTP request consists of a command line, one or more header lines, a blank line, and optionally, input data to the command. In the case of the above URL, there is no input data, and so the request will look something like Figure 11-3(a).

```
GET  /docs/books/tutorial/jdbc/index.html  HTTP/1.1
Host: mycomputer.somewhere.edu
```

(a) The HTTP request from the browser

```
HTTP/1.1  200  OK
Date: Saturday, 1 Jan 2000 11:12:13 GMT
Content-Type: text/html
Content-Length: 56789

<html>
.... (lots of HTML code omitted)
</html>
```

(b) The HTTP response from the web server

Figure 11-3: The interaction between a browser and web server

The command line specifies the requested operation (i.e., GET), the path to the document, and an indication of which HTTP version the browser is using. The single header line in the figure identifies the client machine. Additional header lines are possible, such as to identify the browser version, specify the desired format of the output data, and send cookie information.

After the web server processes the request, it sends an *HTTP response* to the browser. The response has basically the same structure as a request: It consists of a status line, one or more header lines, a blank line, and the output data. The response to our example request will look something like Figure 11-3(b).

The output data returned by the web server can be in any format: XML, HTML, JPEG, and so on. The *Content-Type* header identifies the format so that the browser can display it correctly.

11.2.2 Dynamic requests

Suppose now that you type the following URL into your browser:

```
http://www.imdb.com:80/find?s=tt&q=jaws
```

This URL causes your browser to contact the IMDb movie website and search for a movie titled “jaws”. Let’s examine what is going on.

The portion of the URL that specifies the document is “`/find?s=tt&q=jaws`”. Unlike the previous URL, this document is not an HTML file whose contents are to be returned. Instead, it is a call to a program on the web server. The program name is “find”, which is located in the server’s root directory. Two arguments are supplied as input to the program. The argument named *s* has value “tt”, and the argument named *q* has value “jaws”. In this program, the first argument specifies the type of search – in particular, the value “tt” says to search by title. The second argument gives the search string.

The portion of the URL that specifies the arguments is called the *query string*. A query string is represented as a list of *name=value* pairs, separated by the “&” character. Since arguments are named, their order in the query string is irrelevant. The “?” character separates the query string from the program name.

> *The input to a web program is contained in the query-string portion of the URL.*

The called program is responsible for generating a proper HTTP response, which the web server can then return to the browser. For example, the *find* program at the *imdb.com* website generates a response that contains HTML code and a content-type value of “text/html”.

11.2.3 HTML forms

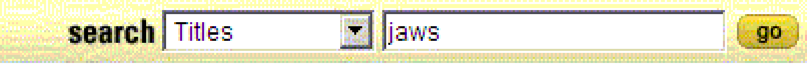
Query strings are an awkward and error-prone way to provide input to a program. There are two main problems:

- The values of the arguments need to be *URL-encoded*. That is, the values cannot contain certain characters (such as spaces and ampersands), and so each such character must be specially encoded. For example, spaces are encoded by the string "%20".
- The names of the arguments used by a program are often obscure, as are their possible values. A notable example is in the above query string, where "`s=tt`" means "search by title".

A common solution to this problem is to have users call the program by submitting a form.

A form is a user-friendly way to create a query string.

For example, there is a form on the *imdb.com* home page that makes it easy for a user to call the *find* program. Figure 11-4(a) contains a screenshot of that form. The form contains three input controls: a dropdown box for selecting the type of search; a text box for specifying the search string; and a submit button. (The bold text "search" is not part of the form.) A user specifies the arguments to the *find* program by selecting a menu item from the dropdown box and typing a search string into the text box. The user then calls the program by clicking on the button. The browser is responsible for creating the proper URL-encoded query string from the values of the two controls.



(a) What the form looks like

```
<form method="get" action="/find">
   <select name="s">
      <option value="all">All</option>
      <option value="tt">Titles</option>
      <option value="nm">Names</option>
      <option value="kw">Keywords</option>
    </select>
   <input type="text" name="q" size="28">
   <input type="submit" value="go">
</form>
```

(b) The HTML code for the form (slightly edited)

Figure 11-4: A form from the *imdb.com* website

Figure 11-4(b) shows the HTML code for this form.[†] Forms use the bracketed <form> and </form> tags. The <form> tag has two important attributes: *method* and *action*. The *action* attribute specifies the name of the program to be called; the *method* attribute specifies the command to be used in the HTTP request. (In addition to GET, there is also a POST command, which we shall consider shortly.)

When the form's submit button is pressed, the browser constructs a query string by extracting the name and value of each control in the form. For example in Figure 11-4, the dropdown box has name "s" and value "tt", and the text box has name "q" and value "jaws". The query string created by the browser is thus

```
s=tt&q=jaws
```

Note that this query string is identical to the string discussed in the previous section, which means that the request that the form sends to the web server is identical to the request that would get created if you typed in the URL yourself. In other words, this form is just a convenience for the user – the web server cannot tell if a request is coming from the form or not.

When you type a URL into a browser, it always uses the GET command to send the HTTP request to the web server. The reason is that you are asking the browser to retrieve information from the web server, and that is the purpose of the GET command.

The other important HTTP command is POST, whose purpose is to change the state of the application. Examples of interactions that should use POST include:

- inserting, deleting, or modifying a record in the underlying database;
- adding an item to an online shopping cart;
- identifying yourself to the server via a username and password.

There is only one way to get a browser to send a POST command; you must use a form whose *method* attribute has the value "post".

The primary difference between the POST and GET commands is the location of the query string. In a GET command, the query string is passed to the web server via the URL. In a POST command, the query string is placed in the input-data section of the HTTP request. As a result, the query string of a POST request can be arbitrarily long (which is useful for uploading the contents of a file), and will be invisible to casual users (which is useful for passing sensitive values such as passwords).

Forms that send sensitive input to the web server
should always use the POST command.

So far, we have discussed how the browser processes a form. We now need to discuss how to create various controls inside a form.

[†] The code has been edited slightly from what appears on the *imdb.com* website, for compactness and clarity.

Text boxes

A text box control is created by an *input* tag whose *type* attribute has the value "text". In Figure 11-4(b), the tag looks like this:

```
<input type="text" name="q" size="28">
```

The *name* attribute specifies the name of the control, and the *size* attribute specifies its width. The value of the control is the text that the user types into the box.

Check boxes

A check box control is created by an *input* tag whose *type* attribute has the value "checkbox". A check box gets toggled on/off each time the user clicks on it. Each check box on a form can be selected independently of the others. The following tag produces a checkbox named *xxx*:

```
<input type="checkbox" name="xxx" value="yes">
```

The check box will have its specified value when it is checked, and will have no value when it is unchecked. A browser will only add a control to the query string if it has a value. Consequently, checkboxes that are unchecked will not appear in the query string.

Radio Buttons

A radio button control is created by an *input* tag whose *type* attribute has the value "radio". All radio buttons having the same name are in the same group. If a button is checked, all other buttons in its group are unchecked. The following tags create a group of three radio buttons named *yyy*.

```
<input type="radio" name="yyy" value="v1">
<input type="radio" name="yyy" value="v2">
<input type="radio" name="yyy" value="v3">
```

As with check boxes, the browser only adds the value of the selected radio button to the query string. Thus we can tell which button was selected by looking at the value for *yyy* in the query string.

Dropdown Boxes

A dropdown box is created by the *select* and *option* tags. Consider for example the code for the dropdown box in Figure 11-4(b). The *select* tag specifies that the name of the control is "s". The *option* tags specify the four menu items in the dropdown list. Each *option* tag specifies two things: the string that will be displayed for that menu item; and the value of the control if the item is selected. These two things are specified in different ways. The tag's *value* attribute specifies the value of the control, whereas the string to be displayed appears between the opening and closing *option* tags.

If we look back to the screenshot of Figure 11-4(a), we can see that the second menu item was selected – that is, the selected menu item is the one having the display string "Titles". The value of this menu item, however, is "tt". Since the name of the control is "s", the browser will add "*s=tt*" to the query string.

Submit Buttons

Every form should have exactly one submit button. A submit button control is created by an *input* tag whose *type* attribute has the value "submit". The *value* attribute specifies the text that will be displayed on the button. Pressing the button causes the browser to call the program listed in the *action* attribute of the form.

11.3 Basic Servlet Programming

Now that we have an idea of how the web server processes HTTP requests, we turn to the issue of how the web server executes programs. Although the programs can be written in any language, we shall restrict our attention to Java. A webserver program written in Java is called an *HTTP servlet*. Java has the ability to define other kinds of servlet, but they are essentially never used. Thus we shall use the term "servlet" to mean "HTTP servlet".

A servlet is Java code that can be executed by a web server.

11.3.1 The servlet APIs

The Java package *javax.servlet.http* contains an abstract class *HttpServlet* that implements the basic servlet functionality. Figure 11-5 gives the relevant APIs[†].

[†] These servlet classes are part of the Java EE distribution, and are therefore not distributed with the Java 6 SDK. The site *java.sun.com/products/servlet* contains directions for downloading both the classes and their documentation.

HttpServlet

```
void doGet(HttpServletRequest req, HttpServletResponse resp)
                           throws ServletException, IOException;
void doPost(HttpServletRequest req, HttpServletResponse resp)
                           throws ServletException, IOException;
```

HttpServletRequest

```
String      getParameter(String name);
String      getLocalName();
```

HttpServletResponse

```
void        setContentType(String type);
PrintWriter getWriter() throws IOException;
```

Figure 11-5: The API for the basic servlet methods

An *HttpServlet* object contains code for the methods *doGet* and *doPost*, but that code does nothing but throw an exception. Our job, when writing a servlet, is to override one or both of these methods. That is, we implement a servlet by writing a class that extends *HttpServlet* and contains code for *doGet* and/or *doPost*.

A web server will create exactly one object for each servlet class, which is (not unexpectedly) called its *servlet*. When the web server receives a request to execute a servlet, it creates a new thread and calls one of the servlet's methods in that thread. More specifically: if the HTTP command is GET, then the web server executes the servlet's *doGet* method in a new thread; if the command is POST, the web server executes the servlet's *doPost* method in the new thread.

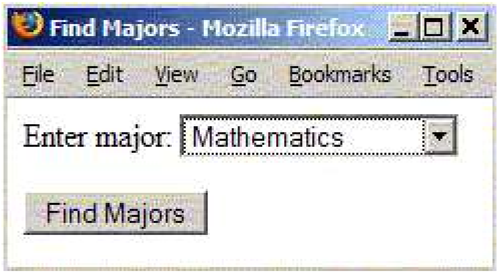
The web server passes two objects to the *doGet* and *doPost* methods. The first object is of type *HttpServletRequest*, and encodes all of the information that the servlet needs to know about the request. The method *getParameter* allows the program to access the query string; given a name, the method returns its value. And the method *getLocalName* returns the domain name of the computer making the request.

The second argument to *doGet* and *doPost* is an object of type *HttpServletResponse*. This object contains methods that the servlet can call to construct the HTTP response that gets sent back to the browser. The method *setContentType* specifies the value of the HTTP Content-type header; in most cases, the servlet will set value to be "text/html". The method *getWriter* returns a *Writer* object. Whatever the servlet sends to this writer will get added to the body of the response.

11.3.2 Writing servlet code

The difference between a servlet and a stand-alone application is primarily in the way that they process input and output. In terms of the view/model distinction of Chapter 9, we would say that a servlet version of a stand-alone application uses the same domain model, but has a different view.

For an example, let’s see how to rewrite the *FindMajors* demo client of Figure 8.15 as a servlet. The first thing to do is design the view. We’ll need two pages: one page to display a user-input form, and one page to display the results. Figure 11-6 depicts a simple input-form page and its HTML code.



Enter major: Mathematics

(a) The browser’s display of the form

```
<html>
<head><title>Find Majors</title></head>
<body>
<form method="get" action="/simpledb/ServletFindMajors">
Enter major:
  <select name="major">
      <option value="math">Mathematics</option>
      <option value="compsci">Computer Science</option>
      <option value="drama">Drama</option>
  </select>
<p>
<input type="submit" value="Find Majors">
</form>
</body>
</html>
```

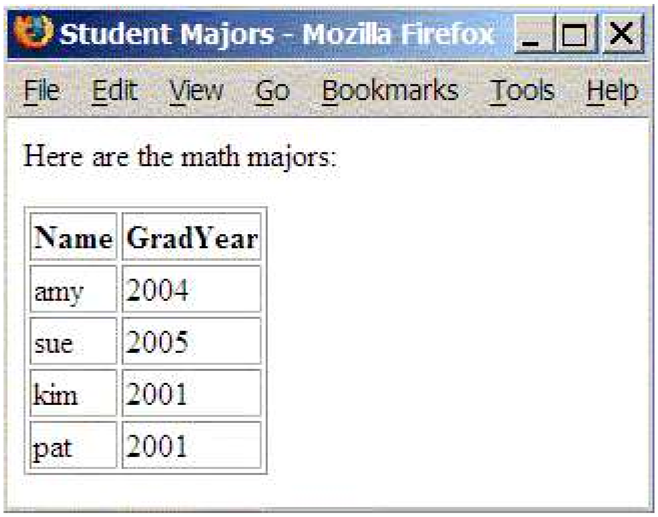
(b) The HTML code for the form

Figure 11-6: The input form for the webserver-based *FindMajors* client

Note that the name of the dropdown-box control is “major”. If the user selects the menu item for the Mathematics department, the browser will create the query string “*major=math*”.

Also note that the form’s action is “/simpledb/ServletFindMajors”. Since the action does not include a machine name, the server assumes that the program is on the same machine as the form. Thus the server will invoke the *ServletFindMajors* program located in the *simpledb* directory of the web server.[†]

Figure 11-7 depicts the servlet's output page and corresponding HTML code. The information about the student majors is displayed as a table, with one student per row.



Here are the math majors:

Name	GradYear
amy	2004
sue	2005
kim	2001
pat	2001

(a) What the browser displays

```
<html>
<head>
<title> Student Majors </title>
</head>
<body>
<P>Here are the math majors:
<P><table border=1>
<tr> <th>Name</th> <th>GradYear</th> </tr>
<tr> <td>amy</td> <td>2004</td> </tr>
<tr> <td>sue</td> <td>2005</td> </tr>
<tr> <td>kim</td> <td>2001</td> </tr>
<tr> <td>pat</td> <td>2001</td> </tr>
</table>
</body>
</html>
```

(b) The corresponding HTML code

Figure 11-7: The output of the webserver-based *FindMajors* client

Once we have designed the view pages, we know what input the servlet will get, and what output it needs to generate. We can then write the code for the *ServletFindMajors* class. This code appears in Figure 11-8.

[†] Actually, the name and location of the servlet may be otherwise, because web servers can be configured to spoof these values. We shall discuss this issue in Section 11.5.

```
public class ServletFindMajors extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
                               throws IOException, ServletException {

      String major = request.getParameter("major");

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<head> <title>Student Majors</title>" </head>");
      out.println("<body>");
      out.println("<P>Here are the " + major + " majors:");

      Connection conn = null;
      try {
         // Step 1: connect to database server
         Driver d = new ClientDriver();
         String url = "jdbc:derby://localhost/studentdb;create=false;";
         conn = d.connect(url, null);

         // Step 2: execute the query
         String qry = "select SName, GradYear "
                    + "from STUDENT, DEPT "
                    + "where DId = MajorId and DName = ?";
         PreparedStatement pstmt = conn.prepareStatement(qry);
         pstmt.setString(1, major);
         ResultSet rs = pstmt.executeQuery();

         // Step 3: loop through the result set
         out.println("<P><table border=1>");
         out.println("<tr> <th>Name</th> <th>GradYear</th> </tr>");
         while (rs.next()) {
            String name = rs.getString("SName");
            int    year = rs.getInt("GradYear");
            out.println("<tr> <td>" + name + "</td> <td>"
                                   + year + "</td> </tr>");
         }
         out.println("</table>");
         rs.close();

      }
      catch(Exception e) {
         e.printStackTrace();
         out.println("SQL Exception. Execution aborted");
      }
      finally {
         out.println("</body> </html>");
         try {
            if (conn != null)
               conn.close();
         }
         catch (SQLException e) {
            e.printStackTrace();
            out.println("Could not close database");
         }
      }
```

```
    }
}
```

Figure 11-8: The code for the *ServletFindMajors* class

Note is that the class implements the *doGet* method but not *doPost*. The rationale is that we only expect the servlet to be called using the GET command. If a browser calls the servlet using POST, the servlet will return an error page.

The first thing that the method *doGet* does is extract its input value from the query string. It then starts constructing the response: it sets the content type, and writes some of the HTML code to the response's writer.

The second thing that *doGet* does is connect to the database and get the desired result set. The important thing to realize about this code is that its JDBC calls are identical to those of the stand-alone client *FindMajors*. In both cases, the programs execute the same query, and print its output table by looping through the result set and writing out each record.[†] The primary difference between the two classes is that the servlet also has to write appropriate HTML tags, so that the browser will format the values correctly.

The moral is this:

> *Although stand-alone and webserver-based applications are very different architecturally, they are no different with respect to database access.*

11.3.3 Servlets that generate forms

A problem with the input form of Figure 11-6 is that the department names are hard-coded into the dropdown box. If departments get added to or deleted from the database, then the form will have to be modified. This problem is unavoidable for forms written in HTML, because they are static and cannot access the database.

The way to get around this problem is to use a servlet to generate the form; see Figure 11-9. The servlet connects to the database server, downloads the current list of department names, and constructs the appropriate HTML code for the form.

[†] Another similarity is that neither program separates the model from the view, as discussed in Chapter 9. Exercise 11.9 asks you to rectify the situation.

```
public class ServletFindMajorsInputForm extends HttpServlet {
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
                           throws IOException, ServletException {

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<head> <title>Find Majors</title> </head>");
      out.println("<body>");
      out.println("<form method=\"get\" "
                      + "action=\"/studentdb/ServletFindMajors\">");
      out.println("Enter major: <select name=\"major\">");

      Connection conn = null;
      try {
         // Step 1: connect to database server
         Driver d = new ClientDriver();
         String url = "jdbc:derby://localhost/studentdb;create=false;";
         conn = d.connect(url, null);

         // Step 2: execute the query
         Statement stmt = conn.createStatement();
         String qry = "select dname from dept";
         ResultSet rs = stmt.executeQuery(qry);

         // Step 3: loop through the result set
         while (rs.next()) {
            String dname = rs.getString("DName");
            out.println("<option value=\"" + dname + "\">"
                                         + dname + "</option>");
         }
         rs.close();
         out.println("</select>");
         out.println("<p>");
         out.println("<input type=\"submit\" value=\"Find Majors\">");
         out.println("</form>");
      }
      catch(Exception e) {
         e.printStackTrace();
         out.println("SQL Exception. Execution aborted");
      }
      finally {
         out.println("</body> </html>");
         try {
            if (conn != null)
               conn.close();
         }
         catch (SQLException e) {
            e.printStackTrace();
            out.println("Could not close database");
         }
      }
   }
}
```

Figure 11-9: The code for the *ServletFindMajorsInputForm* class

Note how this code is remarkably similar to the *ServletFindMajors* servlet: It prints the initial HTML tags; it obtains a recordset from the database server; it loops through that recordset, printing some HTML for each output record; and it prints the remaining HTML tags. The only difference is that it is printing a form instead of a table.

Note also how the servlet needs to generate tags whose attribute values contain double quotes. The Java code therefore needs to escape these double quotes inside its calls to *println*.

11.4 Managing Database Connections

The previous section asserted that stand-alone and webserver-based applications are no different with respect to database access. However, there is actually one difference: Each stand-alone application runs as a separate process, whereas servlets all run as separate threads inside the web server's process. This distinction might not seem like much, but it turns out to be quite important because it affects the way that database connections can be managed.

In Section 9.2.3 we discussed this issue for stand-alone applications. We saw that stand-alone applications were in the following no-win situation: If the application decides to hold onto its connection between transactions, then it is hogging the resource and may keep other users from connecting to the database server; but if the application decides to release its connection between transactions, then its response time slows appreciably each time that it has to reconnect.

The solution given in that section was for each client to use a *connection pool server*. Such a server would maintain a pool of connections. Whenever a stand-alone application needed to connect to the database, it would request a connection from the server; and when the application finished with the connection, it would return the connection to the pool. This technique allows many applications to share a small number of connections.

The problem with using a connection pool server is that the server would most likely be on a different machine from the client, which means that each client would have to make a network connection in order to obtain a database connection. Even though a network connection is less expensive to get than a database connection, there is still a nontrivial cost involved.

The servlet architecture avoids this problem. Since all servlets run in the same process as the web server, the webserver can manage the connection pool. The connection-pool service is thus local to each servlet, which reduces its cost considerably.

In order to take advantage of connection pooling, a servlet must no longer get its connection from the database driver; instead, it must get the connection from the web server. The web server uses a customized *DataSource* object for this purpose. Recall from Section 8.2.1 that a *DataSource* object encapsulates everything about how to connect to the server. The client does not know any of the connection details, because it obtains its connection by calling the zero-argument method *getConnection*. The trick, therefore, is for the web server to create a *DataSource* object whose *getConnection* and *close* methods access the connection pool; it then gives this object to its servlets.

Servlets obtain the customized *DataSource* object via a *name server*. When the web server first starts up, it creates the *DataSource* object and publishes it with a name server. When a servlet needs to use the database, it retrieves a copy of the *DataSource* object from that name server.

The Java name server interface is called *JNDI* (for *Java Naming* and *Directory Interface*). Figure 11-10 shows how we would rewrite step 1 of the servlet code in Figure 11-8 (or Figure 11-9) to use JNDI methods.

```
// Step 1: connect to database server
Context root  = new InitialContext();
String  path  = "java:comp/env/jdbc/studentdb";
DataSource ds = (DataSource) root.lookup(path);
conn = ds.getConnection();
```

Figure 11-10: Rewriting Step 1 of *ServletFindMajors*
to use a webserver-supplied data source

A JNDI name server publishes objects in a tree-structured directory. Figure 11-10 assumes that the webserver has stored the *DataSource* object in the path "java:comp/env/jdbc/studentdb", starting from the root of the tree. The *InitialContext* object in the code denotes the root of the tree, and the method *lookup* returns the object stored at the specified path.[†]

11.5 Configuring a Servlet Container

Learning about servlets is more interesting if you can actually run some code. If you have access to a servlet-enabled webserver, then you could use that. But it is also very easy to download and run a webserver on your own personal machine.

The webserver we shall use is called *Tomcat*. Download and install the latest version from the *tomcat.apache.org* website. After installation, make sure that you can run the webserver and access its example servlets, as described in its documentation.

11.5.1 Configuring a web application

We want to add our own content to the web server. In doing so, we will need to add new directories and files, and change existing files. By default, the Tomcat server will not

[†] You might be asking yourself why the web server would use such an obscure pathname. The reason is largely due to convention, as will be discussed in the next section. In general, a JNDI server might be used for many purposes besides servlets, and this particular pathname keeps servlet names separate from other names.

recognize any of these changes while it is running. Therefore you must remember to restart the server each time that you change a file or add a new one. (Alternatively, you can configure Tomcat to recognize changes dynamically; see its installation guide.)

A web server partitions the pages in its website into *web applications (*or *webapps)*. For example, Tomcat comes with a demo webapp called "examples". This webapp contains all of the demo servlets and related HTML files. Similarly, we shall create a webapp for the university database called "studentdb".

The contents of each webapp is stored as a subdirectory of the Tomcat *webapps* directory. So if we want to create our *studentdb* webapp, the first thing we need to do is create a subdirectory named "studentdb". This subdirectory is called the webapp's *root directory*.

The HTML files belonging to a webapp are placed directly in its root directory. For example, we can store the static input form of Figure 11-6 in the file *findMajorsInputForm.html*, and place this file in the *studentdb* subdirectory. If we do so, then we can access this form by typing the following URL into our browser:

```
http://localhost:8080/studentdb/findMajorsInputForm.html
```

Note that the pathname of the file consists of the root directory of the webapp, followed by the name of the html file. Note also that this URL assumes that Tomcat is listening on port 8080, which is the port used by the default configuration.

In order for a webapp to process servlets, its root directory must contain a subdirectory called WEB-INF. That subdirectory should contain three important items:

- a directory called *classes*. This is where you put the class files for each servlet, as well as auxiliary non-servlet class files. The web server will place this directory on its CLASSPATH. Therefore if your classes belong to packages, then they will need to be in subdirectories corresponding to the package name.
- a directory called *lib*. This is where you can put jar files of auxiliary classes that your servlets use. For example in the *studentdb* webapp, we could either place all of the JPA model classes of Chapter 9 in the *classes* directory, or we could archive them into a jar file and place this file in the *lib* directory.
- a file called *web.xml*. This file is called the *deployment descriptor* for the webapp, and will be discussed shortly.

The webapp's *lib* directory should only contain webapp-specific classes. If you have jar files that are relevant to all webapps (for example, JDBC driver jar files), then you should put them in Tomcat's global *lib* directory.

The deployment descriptor for each webapp contains servlet-specific configuration information. Although there are several kinds of configuration specifications, we are interested in just one of them: configuring how servlets are called. Figure 11-11 contains a simple *web.xml* file for the *studentdb* webapp.

```
<web-app>
   <servlet>
      <servlet-name>FindMajors</servlet-name>
      <servlet-class>ServletFindMajors</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>FindMajors</servlet-name>
      <url-pattern>/ServletFindMajors</url-pattern>
   </servlet-mapping>

   <servlet>
      <servlet-name>FindMajorsForm</servlet-name>
      <servlet-class>ServletFindMajorsInputForm</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>FindMajorsForm</servlet-name>
      <url-pattern>/forms/findmajors.html</url-pattern>
   </servlet-mapping>
</web-app>
```

Figure 11-11: The *web.xml* deployment descriptor for the *studentdb* webapp

This file has entries for each of the two servlets developed in this chapter. The *servlet* and *servlet-mapping* elements deal with the naming of each servlet. The *servlet* element associates an internal name for the servlet with its class. This internal name is then used in the *servlet-mapping* element, which assigns a URL to the servlet. In the figure, the URL assigned to the *FindMajors* servlet is the same as its class name, prepended with "/". The URL assigned to the *FindMajorsForm* servlet is "/forms/findmajors.html". When you call a servlet, you append its URL to the URL of the webapp. For example, our two servlets would be accessed from a browser by typing the following URLs:

```
http://localhost:8080/studentdb/ServletFindMajors?major=math
http://localhost:8080/studentdb/forms/findmajors.html
```

The mapping for *FindMajorsForm* illustrates how you can spoof the actual name of the servlet class, the directory where the servlet lives, and even whether the file is a servlet or not. That is, there is no directory named "forms", nor is there a file named "findmajors.html". It is impossible for a user to tell whether the second URL calls a servlet or retrieves an HTML file.

11.5.2 Configuring JNDI resources

Section 11.4 discussed how the webserver can support connection pooling by publishing a *DataSource* object to a JNDI server. This *DataSource* object is known as a *JNDI resource*, because the object is available for any servlet to use.

In Tomcat, you configure a resource by adding a *Resource* element to the web server's global configuration file. This file is called *context.xml*, and is in Tomcat's *conf* directory. Figure 11-12 gives the relevant portion of that file.

```
<!-- This file will be loaded for each web application -->
<Context>
   <Resource name="jdbc/studentdb"  type="javax.sql.DataSource"
             driverClassName="org.apache.derby.jdbc.ClientDriver"
             url="jdbc:derby://localhost/studentdb;create=false;"
             maxActive="100"  maxIdle="30"  maxWait="10000"
             auth="Container"/>
...
</Context>
```

Figure 11-12: Part of the web server's *context.xml* configuration file

The *Resource* element is implemented as an empty tag that has several attributes. The values of these attributes give the webserver enough information to create the object and publish it to the JNDI server.

- The *name* attribute specifies the path in the JNDI tree where the resource will be stored. We shall discuss the rationale for the value "jdbc/studentdb" in the next paragraph.
- The *type* attribute specifies the class or interface of the resource.
- The *driverClassName* attribute tells the webserver which JDBC driver to use.
- The *url* attribute specifies the connection string needed by the driver.
- The *maxActive*, *maxIdle*, and *maxWait* attributes are for connection pooling. They say that the connection pool can have at most 100 active connections and at most 30 idle connections. And a client that has been waiting for a connection for more than 10,000 milliseconds will be automatically rolled back.
- The *auth* attribute specifies whether the resource will be managed by the webserver (the "container"), or the webapp.

The value of the *name* attribute, "jdbc/studentdb", indicates that the resource has the name "studentdb", and is to be stored in the part of the subtree dedicated to JDBC resources. This naming convention is not necessary – we could place the *studentdb* object anywhere in the tree we want – but this is the convention suggested by Sun, it avoids name conflicts, and frankly, there is no reason not to use it.

A quick look back at Figure 11-10 shows that servlets actually access the object using the pathname "java:comp/env/jdbc/studentdb". The prefix, "java:comp/env", is added by the web server for all resources that it manages. Again, this is a naming convention. The subtree "java:comp" is intended to hold the names of Java software components; and the "env" subtree holds environment-related names.

11.6 Chapter Summary

- A *stand-alone application* is code that lives on the client machine and accesses the database server directly. A *webserver-based application* is code that lives on the webserver machine and is executed by a browser's request.

- The web server may process a request by returning the contents of a file (a *static* request), or by executing a program and returning its output (a *dynamic* request). The code that fulfills a dynamic request can access the database. In this case, the browser is a client of the web server, and the web server is a client of the database server.

- When you type a URL into a browser, it sends an *HTTP request* to the specified web server. After the web server processes the request, it sends an *HTTP response* to the browser.

- The portion of the URL that specifies the arguments to a webserver program is called the *query string*. A query string is represented as a list of *name=value* pairs, separated by the "&" character.

- Forcing users to manually type a query string is an awkward and error-prone way to provide input to a program. A better way is to let users submit an HTML form. A form contains controls and a submit button. When the button is pressed, the browser constructs a query string from the name and value of each control in the form.

- A webserver program written in Java is called a *servlet*. When the web server receives a request to execute a servlet, executes one of the servlet's methods in a new thread. The method will be either *doGet* or *doPost*, depending on the HTTP command embedded in the request.

- The *doGet* and *doPost* methods take two arguments: an *HttpServletRequest* object, which contains information about the request, and an *HttpServletResponse* object, which allows the servlet to construct its HTML response.

- The difference between a servlet and a stand-alone application is primarily in how they process input and output. In particular, a servlet version of a stand-alone application uses the same model, but has a different view. Consequently, both kinds of application interact with the database server in almost exactly the same way.

- The one difference is that a webserver-based application can use connection pooling effectively, because all of the servlets execute in the same process on the same machine. A web server can implement connection pooling by creating a *DataSource* object whose *getConnection* method uses the connection pool, and publishing that object in a JNDI name server. Each servlet can then retrieve a copy of the *DataSource* object from the name server.

- A web server divides the pages on its website into *web applications (*or *webapps)*. A web server contains a directory for each webapp; this directory has a specific structure, in which the servlets, HTML files, and supporting classes are placed. Each webapp also has a deployment descriptor, which is used to provide configuration information to the web server.

11.7 Suggested Reading

There are several websites that serve as a reference for HTML forms, such as *http://www.htmlcodetutorial.com/forms*. Instead of coding forms directly in HTML, we could use a client-side scripting language, such as JavaScript or ActionScript [Deitel et al. 2003].

We have paid minimal attention to the issue of designing an effective website. As we indicated in Section 11.3.2, you cannot even think about writing servlets until you have completely thought out the content of each page and designed the flow of control from one page to the next. The book [Rosenfeld and Morville 2006] is a good introduction to the design process and the issues involved.

We have described only the basic servlet functionality. For example, a sophisticated website would want to take advantage of features such as sessions and cookies. Moreover, servlets are a somewhat awkward way to develop web pages. A more popular technique is to write a web page using HTML, with additional tags that contain Java code; such a file is called a *JSP* file (for *Java Server Pages*). A web server executes a JSP file by compiling it into a servlet and then executing the servlet. The book [Hall and Brown 2004] gives a comprehensive coverage of these topics. An online tutorial for servlets and JSP is also available in Chapters 11-16 of the Sun J2EE tutorial, which can be downloaded from *http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html.*

Servers that can execute code for clients are often known as *middleware*. There are many kinds of middleware servers in addition to web servers. The Java EE environment is an umbrella for many such servers, such as web servers, transaction servers, message servers, and so on. The online tutorial mentioned above is a good introduction to the middleware provided by J2EE. Many of the ideas we have seen for web servers apply to middleware in general. Of course, the more kinds of servers we have available, the more design possibilities there are. The book [Fowler 2003] explains the various choices and tradeoffs that an application designer can make.

11.8 Exercises

CONCEPTUAL EXERCISES

11.1 The Tomcat configuration of Section 11.5 accessed the Derby database server exclusively.
a) Suppose you wanted to change the servlets in the *studentdb* webapp to be able to use the SimpleDB driver instead. What would you do?
b) Suppose you wanted to change the Tomcat connection pool to use SimpleDB instead of Derby. What would you do?
c) Is it possible to have two connection pools, one for each server? How would you do it?

d) Explain why is it not possible to have a single pool that manages connections for both servers at the same time.

11.2 Explain why you might want to configure Tomcat so that it publishes several data sources for the Derby server. How would you revise the configuration file of Figure 11-12?

11.3 Suppose we decide to change the *studentdb* webapp so that the *ServletFindMajors* servlet is accessed by the following URL:

```
http://localhost:8080/studentdb/findwhere?major=math
```

a) Explain how the *studentdb* deployment descriptor needs to change.
b) Explain how the forms that call that servlet need to change.

PROGRAMMING EXERCISES

11.4 Write an HTML form whose only control is a submit button, and whose action is the URL of some HTML page (not a servlet).
a) What is the effect of pressing the button? Is there another way to do the same thing in HTML?
b) Add a few controls to that form. What happens when you press the button? What does the submitted URL look like? Is it what you expected?

11.5 Revise the HTML form of Figure 11-6:
a) to use a text box instead of a dropdown box.
b) to use radio buttons instead of a dropdown box.

11.6 Revise the servlet of Figure 11-9:
a) to generate a text box instead of a dropdown box.
b) to generate radio buttons instead of a dropdown box.

11.7 Consider the *StudentInfo* stand-alone application of Section 9.1. We want to rewrite it as a webserver-based application.
a) Design web pages for input and output. (HINT: An HTML page can contain more than one form.)
b) Write a servlet for each form that you use. The servlet should connect to the database directly.
c) Configure your version of Tomcat so that it publishes a connection-pool data source. Then rewrite the servlet so that it connects to the database via this data source.

11.8 Consider the *JPAStudentInfo* stand-alone application of Section 9.3.
a) Rewrite it as a webserver-based application. (HINT: The *persistence.xml* file still needs to be in a directory named META-INF, and this directory still needs to be in the same directory as the model classes.)
b) Explain why your code cannot take advantage of Tomcat's connection pooling.

11.9 Rewrite the *ServletFindMajors* program of Figure 11-8 so that it uses the JPA domain model of Chapter 9 to separate the model from the view.

11.10 Consider the stand-alone demo applications *StudentMajor*, *ChangeMajor*, and *SQLInterpreter*. Rewrite each of them to be a webserver-based application.

11.11 Create a webserver-based application that allows users to modify the contents of the STUDENT table.
a) Design the web pages. You should have a page for inserting a new student, deleting a selected student, and modifying the fields of an existing student. The forms should be as easy to use as possible. For example, you should provide a dropdown box for selecting existing students.
b) Write the servlet for each page.
c) Add your files to the *studentdb* webapp. Modify its deployment descriptor appropriately.

11.12 Create a webserver-based application for students to register for courses. Design the web pages, write the necessary servlets, and modify the deployment descriptor appropriately.

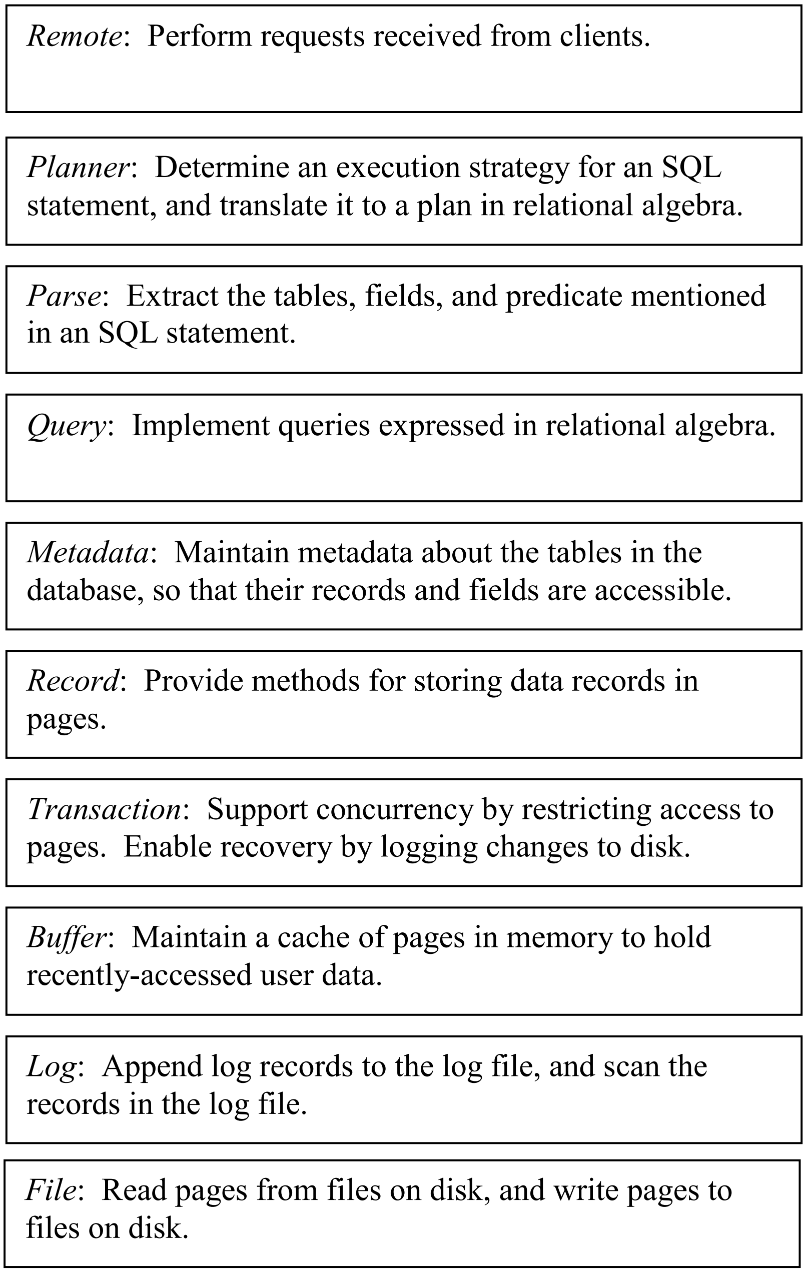
PART 3
Inside the Database Server

Now that we have studied how to use a database system, we move to the question of how database systems are built. A good way to understand how a modern database system works is to study the innards of a real database system. The SimpleDB database system was built exactly for this purpose. The basic system, which we shall examine in Part 3, consists of about 3,500 lines of Java code, organized into 85 classes and 12 packages; and the efficiency optimizations of Part 4 are half again as large. Although that might seem like a lot of code to wade through, we'll see that SimpleDB is an extremely bare-bones system, which uses the simplest possible algorithms and has only the most necessary functionality. Commercial database systems have the same structure, but use sophisticated algorithms, have robust error checking, and numerous bells and whistles. As a result, such systems are significantly larger. Derby, for example, has over 100 times more code than SimpleDB, and Oracle is even larger.

In this part of the text, we break the database server into ten layered components. Each component knows only about the layers below it, and provides services to the layers above it. The picture on the following page illustrates the functionality of each component.

A good way to illustrate the interaction between these components is to trace the execution of an SQL statement. For example, suppose that a JDBC client submits an SQL deletion command to the database server, via the method *executeUpdate*. The following actions occur:

- The remote component implements the *executeUpdate* method. That method creates a new transaction for the statement, and passes the SQL string to the planner.
- The planner sends the SQL string to the parser, which parses it and sends back a *DeleteData* object. This object contains the name of the table mentioned in the statement and the deletion predicate. The planner uses the parser data to determine a *plan* for the statement. This plan represents a relational algebra query for finding the records to be deleted. It sends the plan to the query processor.



Remote: Perform requests received from clients.

Planner: Determine an execution strategy for an SQL statement, and translate it to a plan in relational algebra.

Parse: Extract the tables, fields, and predicate mentioned in an SQL statement.

Query: Implement queries expressed in relational algebra.

Metadata: Maintain metadata about the tables in the database, so that their records and fields are accessible.

Record: Provide methods for storing data records in pages.

Transaction: Support concurrency by restricting access to pages. Enable recovery by logging changes to disk.

Buffer: Maintain a cache of pages in memory to hold recently-accessed user data.

Log: Append log records to the log file, and scan the records in the log file.

File: Read pages from files on disk, and write pages to files on disk.

The components of SimpleDB

- The query processor creates a *scan* from this plan. A scan contains the runtime information needed to execute the plan. To create the scan, the query processor obtains the schema of the affected table from the metadata manager. The query processor then creates a record manager corresponding to this schema.
- The query processor then iterates through the scan. During the iteration, it will examine each record and determine if it satisfies the deletion predicate. Each time it needs to access a record from the table, it calls the record manager. If a record needs to be deleted, the query processor tells the record manager to do so.
- The record manager knows how the records are stored in the file. When the query processor asks it for the value of a record, it determines which block contains the record, and calculates the offset of the desired value within that block. It then asks the transaction manager to retrieve the value at that location of the block.
- The transaction manager checks its lock table to make sure that another transaction is not using the block. If not, the transaction manager locks the block, and calls the buffer manager to get the value.
- The buffer manager keeps a cache of blocks in memory. If the requested block is not in the cache, the buffer manager chooses a page from the cache and contacts the file manager. It asks the file manager to write the current contents of that page to disk, and to read the specified block into that page.
- The file manager reads and writes a block from disk into a memory page, on request.

Some variation of this architecture is used by most every commercial relational database system. The rationale for this architecture is heavily influenced by the nature of disk I/O. Thus in this part of the book, we begin from the lowest level – the disk – and work our way up through the successive levels of database software.

Object-oriented developers often refer to an object that calls methods from some other class a *client* of that class. This use of the term "client" is similar to the client-server usage of Part 2 – in each case, the client object makes a request of an object in another class. The difference is that in the client-server case, the request occurs over the network. In the remainder of this book, we shall use "client" to refer to the object-oriented aspect of the term, and "JDBC client" to refer to the client-server aspect.

12

DISK AND FILE MANAGEMENT

and the package simpledb.file

In this chapter we examine how a database system stores its data on physical devices such as disks and flash drives. We investigate the properties of these devices, and consider techniques (such as RAID) that can improve their speed and reliability. We also examine the two interfaces that the operating system provides for interacting with these devices – a block-level interface and a file-level interface – and discuss which interface is most appropriate for a database system. Finally, we consider the SimpleDB file manager in detail, studying its API and its implementation.

12.1 Persistent Data Storage

The contents of a database must be kept *persistent*, so that the data will not be lost if the database system or the computer goes down. This section examines two particularly useful hardware technologies: *disk drives* and *flash drives*.

Disk drives and flash drives are common examples of persistent storage devices.

Flash drives are not as widespread as disk drives; however, their importance will increase as their technology matures. We begin with disk drives, and consider flash drives afterwards.

12.1.1 Disk drives

A *disk drive* contains one or more rotating *platters.* A platter has concentric *tracks*, and each track consists of a sequence of bytes. Bytes are read from (and written to) the platter by means of a movable arm with a read/write head. The arm is positioned at the desired track, and the head can read (or write) the bytes as they rotate under it. Figure 12-1 depicts the top view of a one-platter disk drive. Of course, this figure is not drawn to scale, because a typical platter has many thousands of tracks.

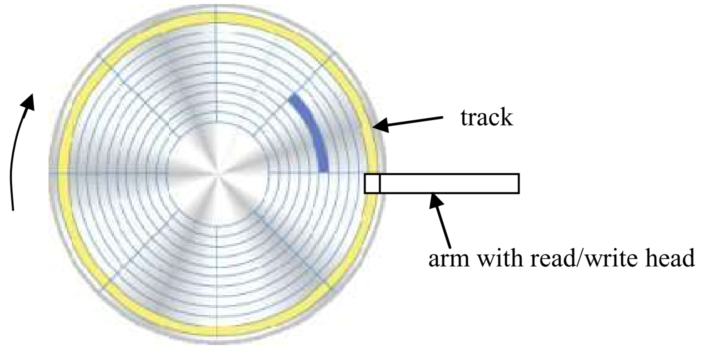


Figure 12-1: The top view of a one-platter disk drive

Modern disk drives typically have multiple platters. For space efficiency, pairs of platters are usually joined back-to-back, creating what looks like a two-sided platter; but conceptually, each side is still a separate platter. Each platter has its own read/write head. These heads do not move independently; instead, they are all connected to a single *actuator*, which moves them simultaneously to the same track on each platter. Moreover, only one read/write head can be active at a time, because there is only one datapath to the computer. Figure 12-2 depicts the side view of a multi-platter disk drive.

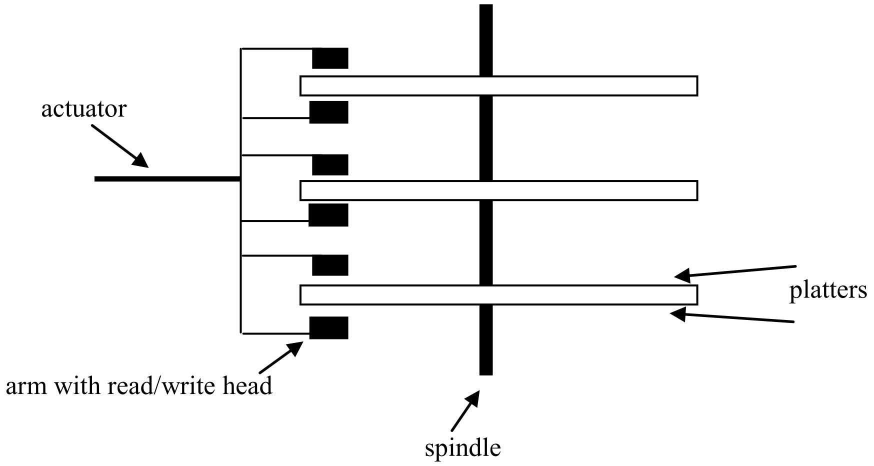


Figure 12-2: The side view of a multi-platter disk drive

We now consider how to characterize the performance of a disk drive.

The general performance of a disk drive can be measured by four values: its capacity, rotation speed, transfer rate, and average seek time.

The *capacity* of a drive is the number of bytes that can be stored. This value depends on the number of platters, the number of tracks per platter, and the number of bytes per track. Given that the platters tend to come in more or less standard sizes, manufacturers increase capacity primarily by increasing the density of a platter – that is, by squeezing more tracks per platter and more bytes per track. Platter capacities of over 40GB are now common.

The *rotation speed* is the rate at which the platters spin, and is usually given as revolutions per minute. Typical speeds range from 5,400rpm to 15,000rpm.

The *transfer rate* is the speed at which bytes pass by the disk head, to be transferred to/from memory. For example, an entire track's worth of bytes can be transferred in the time it takes for the platter to make a single revolution. The transfer rate is thus determined by both the rotation speed and the number of bytes per track. Rates of 100MB/sec are common.

The *seek time* is the time it takes for the actuator to move the disk head from its current location to a requested track. This value depends on how many tracks need to be traversed. It can be as low as 0 (if the destination track is the same as the starting track) and as high as 15-20msec (if the destination and starting tracks are at different ends of the platter). The average seek time usually provides a reasonable estimate of actuator speed. Average seek times on modern disks are about 5 msec.

Consider the following example.

A 4-platter disk drive spins at 10,000rpm with an average seek time of 5msec. Each platter contains 10,000 tracks, with each track containing 500,000 bytes. We can calculate the following values[†].

The capacity of the drive:

```
       500,000 bytes/track x 10,000 tracks/platter
                           x 4 platters/drive
     = 20,000,000,000 bytes, or approximately 20GB
```

The transfer rate:

```
       500,000 bytes/revolution
             x 10,000 revolutions/60 seconds
     = 83,333,333 bytes/second or approximately 83MB/sec
```

12.1.2 Accessing a disk drive

A *disk access* is a request to read some bytes from the disk drive into memory, or to write some bytes from memory to disk. These bytes must be on a contiguous portion of a track on some platter.

[†] Technically, 1KB = 1024 bytes, 1MB = 1,048,576 bytes, and 1GB = 1,073,741,824 bytes. For convenience, we round them down to one thousand, one million, and one billion bytes, respectively.

The disk drive executes a disk access in three stages:

- It moves the disk head to the specified track. As we have seen, this time is called the *seek time*.
- It waits for the platter to rotate until the first desired byte is beneath the disk head. This time is called the *rotational delay*.
- As the platter continues to rotate, it reads each byte (or writes each byte) that appears under the disk head, until the last desired byte appears. This time is called the *transfer time*.

The time required to execute a disk access is the sum of the seek time, rotational delay, and transfer time.

Note that each of these times is constrained by the mechanical movement of the disk. Mechanical movement is significantly slower than electrical movement, which is why disk drives are so much slower than RAM. The seek time and rotational delay are especially annoying. These two times are nothing but overhead that every disk operation is forced to wait for.

Calculating the exact seek time and rotational delay of a disk access is impractical, because it requires knowing the previous state of the disk. Instead, we estimate these times by using their average. We already know about the average seek time. The average rotational delay is easily calculated. The rotational delay can be as low as 0 (if the first byte just happens to be under the head), and as high as the time for a complete rotation (if the first byte just passed by the head). On the average, we will have to wait ½ rotation until the platter is positioned where we want it. Thus the average rotational delay is half of the rotation time.

The transfer time is also easily calculated, by using the transfer rate. In particular, if the transfer rate is b_1 bytes/second and we are transferring b_2 bytes, then the transfer time is b_2/b_1 seconds.

Continuing the above example, we can estimate the cost of some disk accesses:

Consider the disk drive spinning at 10,000rpm, having an average seek time of 5 msec and a transfer rate of 83 MB/sec.

Average rotational delay:

```
      60 seconds/minute x 1 minute/10,000 revolutions
                        x ½ revolution
    = 0.003 seconds or 3 msec
```

Transfer time for 1 byte:

```
      1 byte x 1 second/83,000,000 bytes
    = 0.000000012 seconds = 0.000012 msec
```

Transfer time for 1,000 bytes:

```
      1,000 bytes x 1 second/83,000,000 bytes
    = 0.000012 seconds = 0.012 msec
```

Estimated time to access 1 byte:

```
      5 msec (seek time) + 3 msec (rotational delay)
                         + 0.000012 msec (transfer time)
    = 8.000012 msec
```

Estimated time to access 1,000 bytes:

```
      5 msec (seek time) + 3 msec (rotational delay)
                         + 0.012 msec (transfer time)
    = 8.012 msec
```

Note that the estimated access time for 1,000 bytes is essentially the same as for 1 byte. In other words, it makes no sense to access a few bytes from disk. In fact, you couldn't even if you wanted to. Modern disks are built so that each track is divided into fixed-length *sectors*; a disk read (or write) must operate on an entire sector at a time. The size of a sector may be determined by the disk manufacturer, or it may be chosen when the disk is formatted. A typical sector size is 512 bytes.

12.1.3 Improving disk access time

Because disk drives are so slow, several techniques have been developed that can help improve access times. We shall consider three techniques: disk caches, cylinders, and disk striping.

Disk Caches

A *disk cache* is memory that is bundled with the disk drive, and is usually large enough to store many tracks worth of data. Whenever the disk drive reads a sector from disk, it saves the contents of that sector in its cache; if the cache is full, the new sector replaces an old sector. When a sector is requested, the disk drive checks the cache. If the sector happens to be in the cache, it can be returned immediately to the computer without an actual disk access.

Suppose that an application requests the same sector more than once in a relatively short period. Then the first request will bring the sector into the cache and subsequent requests will retrieve it from the cache, thereby saving on disk accesses. However, this feature is not particularly important for a database system, because the system is already doing the same form of caching (as we shall see in Chapter 13). If a sector is requested multiple times, the database server will find the sector in its own cache and not even bother to go to the disk.

Here is the real reason why a disk cache is valuable. Suppose that the disk never reads a single sector at a time. Instead of reading just the requested sector, the disk drive reads the entire track containing that sector into the cache, in the hope that the other sectors of the track will be requested later. The point is that reading an entire track is not that much more time-consuming than reading a single sector. In particular there is no rotational delay, because the disk can read the track starting from whatever sector happens to be under the read/write head, and continue reading throughout the rotation. Compare the access times:

Time to read a sector = seek time + ½ rotation time + sector rotation time
Time to read a track = seek time + rotation time

That is, the difference between reading a single sector and a track full of sectors is less than half the disk rotation time. So if the database system happens to request just one other sector on the track, then reading the entire track into the cache will have saved time.

The real value of a disk cache is that it allows the disk to pre-fetch the sectors on a track.

Cylinders

The database system can improve disk access time by storing related information in nearby sectors. For example, the ideal way to store a file is to place its contents on the same track of a platter. This strategy is clearly best if the disk does track-based caching, because the entire file will be read in a single disk access. But the strategy is good even without caching, because it eliminates seek time – each time another sector is read, the disk head will already be located at the proper track.[†]

Suppose that a file occupies more than one track. A good idea is to store its contents in nearby tracks of the platter, so that the seek time between tracks is as small as possible. An even better idea, however, is to store its contents on the same track of other platters. Since the read/write heads of each platter all move together, all of the tracks having the same track number can be accessed without any additional seek time.

[†] A file whose contents are wildly scattered across different tracks of the disk is said to be *fragmented*. Many operating systems (such as Windows) provide a de-fragmentation utility that improves access time by relocating each file so that its sectors are as contiguous as possible.

The set of tracks having the same track number is called a *cylinder*, because if you look at those tracks, they seem to describe the outside of a cylinder. Practically speaking, a cylinder can be treated as if it were a very large track, because all of the sectors in it can be accessed with zero additional seeks.

Disk Striping

Another way to improve disk access time is to use multiple disk drives.

Suppose you want to store a 40GB database.
Two 20GB drives will always be faster than a single 40GB drive.

The reason that two small drives are faster than one large drive is that they contain two independent actuators, and thus can respond to two different sector requests simultaneously. In fact, if both 20GB disks are always kept busy, then the disk performance will be about twice as fast as with a single 40GB disk. This speedup scales well: in general, N disks will be about N times as fast as a single disk. (Of course, several smaller drives are also more expensive than a single large drive, so the added efficiency comes at a cost.)

Multiple small disks will only be more efficient if they can be kept busy. For example, suppose that one disk contains the frequently-used files, while the other disks contain the little-used, archived files. Then the first disk would be doing all of the work, with the other disks standing idle most of the time. This setup would have about the same efficiency as a single disk, because there is little simultaneity.

So the problem is how to balance the workload among the multiple disks. The database administrator could try to analyze file usage in order to best distribute the files on each disk, but that approach is not practical: It is difficult to do, hard to guarantee, and would have to be continually re-evaluated and revised over time. Fortunately, there is a much better approach, known as *disk striping*.

The disk striping strategy uses a controller to hide the smaller disks from the operating system, giving it the illusion of a single large disk.

The controller maps sector requests on the virtual disk to sector requests on the actual disks. The mapping works as follows. Suppose there are N small disks, each having k sectors. The virtual disk will have N*k sectors; these sectors are assigned to sectors of the real disks in an alternating pattern. Disk 0 will contain virtual sectors 0, N, 2N, etc. Disk 1 will contain virtual sectors 1, N+1, 2N+1, etc. And so on. The term *disk striping* comes from the following imagery: If you imagine that each small disk is painted in a different color, then the virtual disk looks like it has stripes, with its sectors painted in alternating colors.[†] See Figure 12-3.

[†] Most controllers allow a user to define a stripe to be of any size, not just a sector. For example, a track makes a good stripe if the disk drives are also performing track-based disk caching. The optimal stripe size depends on many factors, and is often determined by trial and error.

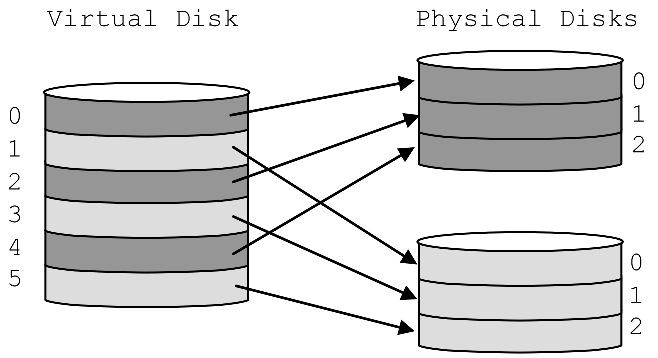


Figure 12-3: Disk striping

Disk striping is effective because it distributes the database equally among the small disks. If a request arrives for a random sector, then that request will be sent to one of the small disks with equal probability. And if several requests arrive for contiguous sectors, they will be sent to different disks. Thus the disks are guaranteed to be working as uniformly as possible.

12.1.4 Improving disk reliability by mirroring

Users of a database expect that their data will remain safe on disk, and will not get lost or become corrupted. Unfortunately, disk drives are not completely reliable. The magnetic material on a platter can degenerate, causing sectors to become unreadable. Or a piece of dust or a jarring movement could cause a read/write head to scrape against a platter, ruining the affected sectors (a "head crash").

The most obvious way to guard against disk failure is to keep a copy of the disk's contents. For example, we could make nightly backups of the disk; when a disk fails, we simply buy a new disk and copy the backup onto it. The problem with this strategy is that we lose all of the changes to the disk that occurred between the time that the disk was backed up and the time when it failed. The only way around this problem is to replicate every change to the disk at the moment it occurs. In other words, we need to keep two identical versions of the disk; these versions are said to be *mirrors* of each other.

As with striping, a controller is needed to manage the two mirrored disks. When the database system requests a disk read, the controller can access the specified sector of either disk, as it chooses. When a disk write is requested, the controller performs the same write to both disks. In theory, these two disk writes could be performed in parallel, which would require no additional time. In practice, however, it is important to write the mirrors sequentially to guard against a system crash. The problem is that if the system crashes in the middle of a disk write, the contents of that sector is lost. So if both mirrors are written in parallel, both copies of the sector could be lost, whereas if the mirrors are written sequentially, then at least one of the mirrors will be uncorrupted.

Suppose that one disk from a mirrored pair fails. The database administrator can recover the system by performing the following procedure:

- Shut down the system.
- Replace the failed disk with a new disk.
- Copy the data from the good disk onto the new disk.
- Restart the system.

Unfortunately, this procedure is not fool-proof. Data can still get lost if the good disk fails while it is in the middle of copying to the new disk. The chance of both disks failing within a couple of hours of each other is small (it is about 1 in 60,000 with today's disks), but if the database is important, this small risk might be unacceptable. We can reduce the risk by using three mirrored disks instead of two. In this case, the data would be lost only if all three disks failed within the same couple of hours; such a possibility, while nonzero, is so remote that it can comfortably be ignored.

Mirroring can coexist with disk striping. A common strategy is to mirror the striped disks. For example, one could store 40GB of data on four 20GB drives: Two of the drives would be striped; the other two would be mirrors of the striped drives. Such a configuration is both fast and reliable. See Figure 12-4.

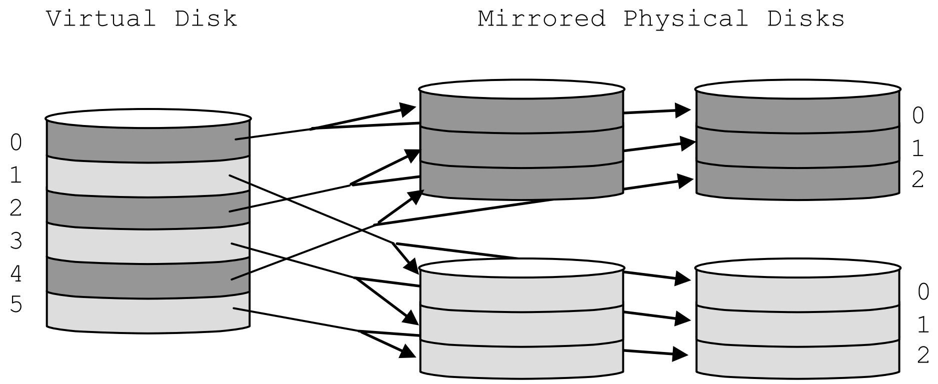


Figure 12-4: Disk striping with mirrors

12.1.5 Improving disk reliability by storing parity

The drawback to mirroring is that it requires twice as many disks to store the same amount of data. This burden is particularly noticeable when disk striping is used – If we want to store 300GB of data using 15 20GB drives, then we will need to buy another 15 drives to be their mirrors. It is not unusual for large database installations to create a huge virtual disk by striping many small disks, and the prospect of buying an equal number of disks just to be mirrors is unappealing. It would be nice to be able to recover from a failed disk without using so many mirror disks.

In fact, there is a clever way to use a single disk to back up any number of other disks. The strategy works by storing *parity* information on the backup disk. Parity is defined for a set S of bits as follows:

- The parity of S is 1 if it contains an odd number of 1's.
- The parity of S is 0 if it contains an even number of 1's.

In other words, if you add the parity bit to S, you will always have an even number of 1's.

Parity has the following interesting and important property: The value of any bit can be determined from the value of the other bits, as long as we also know the parity. For example, suppose that S = {1, 0, 1}. The parity of S is 0 because it has an even number of 1's. Suppose we lose the value of the first bit. Because the parity is 0, the set {?, 0, 1} must have had an even number of 1's; thus we can infer that the missing bit must be a 1. Similar deductions can be made for each of the other bits (including the parity bit).

This use of parity extends to disks. Suppose we have N+1 identically-sized disks. We choose one of the disks to be the parity disk, and let the other N disks hold the striped data. Each bit of the parity disk is computed by finding the parity of the corresponding bit of all the other disks. If any disk fails (including the parity disk), the contents of that disk can be reconstructed by looking, bit by bit, at the contents of the other disks. See Figure 12-5.

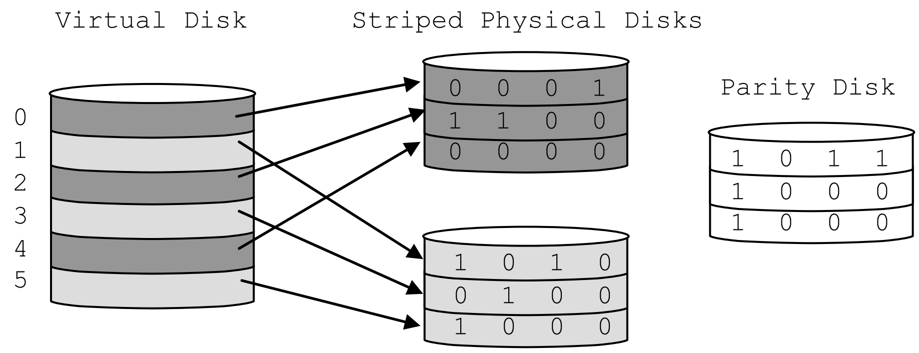


Figure 12-5: Disk striping with parity

The disks are managed by a controller. Read and write requests are handled basically the same as with striping – the controller determines which disk holds the requested sector, and performs that read/write operation. The difference is that write requests must also update the corresponding sector of the parity disk. The controller can calculate the updated parity by determining which bits of the modified sector changed; the rule is that if a bit changes, then the corresponding parity bit must also change. Thus the controller requires four disk accesses to implement a sector write operation: it must read the sector and the corresponding parity sector (in order to calculate the new parity bits), and it must write the new contents of both sectors.

This use of parity information is somewhat magical, in the sense that one disk is able to reliably back up any number of other disks. However, this magic brings two drawbacks with it.

The first drawback to using parity is that a sector-write operation is more time-consuming, as it requires both a read and a write from two disks. Experience indicates that using parity reduces the efficiency of striping by a factor of about 20%.

The second drawback to parity is that the database is more vulnerable to a non-recoverable multi-disk failure. Consider what happens when a disk fails – all of the other disks are needed to reconstruct the failed disk, and the failure of any one of them is disastrous. If the database is comprised of very many small disks (say, around 100) then the possibility of a second failure becomes very real. Contrast this situation with mirroring. Recovering from a failed disk only requires that its mirror not fail, which is much less likely.

12.1.6 RAID

We have considered three ways to use multiple disks:

- *striping* to speed up disk access time;
- *mirroring* and *parity* to guard against disk failure.

All of these strategies use a controller to hide the existence of the multiple disks from the operating system and provide the illusion of a single, virtual disk. The controller maps each virtual read/write operation to one or more operations on the underlying disks. The controller can be implemented in software or hardware, although hardware controllers are more widespread.

These strategies are part of a larger collection of strategies known as *RAID*, which stands for *Redundant Array of Inexpensive Disks*. There are seven RAID levels.

RAID-0 is striping, without any guard against disk failure. If one of the striped disks fails, then the entire database is potentially ruined.

RAID-1 is mirrored striping.

RAID-2 uses bit striping instead of sector striping, and a redundancy mechanism based on error-correcting codes instead of parity. This strategy has proven to be difficult to implement and has poor performance. It is therefore no longer used.

RAID-3 and *RAID-4* use striping and parity. Their difference is that RAID-3 uses byte striping, whereas RAID-4 uses sector striping. In general, sector striping tends to be more efficient because it corresponds to the unit of disk access.

RAID-5 is similar to RAID-4, except that instead of storing all of the parity information on a separate disk, the parity information is distributed amongst the data disks. That is, if there are N data disks, then every Nth sector of each disk holds parity information. This strategy is more efficient than RAID-4, because there is no longer a single parity disk to become a bottleneck. See Exercise 12.5.

RAID-6 is similar to RAID-5, except that it keeps two kinds of parity information. This strategy is therefore able to handle two concurrent disk failures, but needs one more disk to hold the additional parity information.

The two most popular RAID levels are RAID-1 and RAID-5. The choice between them is really one of mirroring vs. parity. Mirroring tends to be the more solid choice in a database installation, first because of its speed and robustness, and second because the cost of the additional disk drives has become so low.

12.1.7 Flash Drives

Disk drives are commonplace in current database systems, but they have an insurmountable drawback – their operation depends entirely on the mechanical activity of spinning platters and moving actuators. This drawback makes disk drives inherently slow compared to electronic memory, and also susceptible to damage from falling, vibration, and other shocks.

Flash memory is a more recent technology that has the potential to replace disk drives. It uses semiconductor technology, similar to RAM, but does not require an uninterrupted power supply. Because its activity is entirely electrical, it has the potential to access data much more quickly than disk drives, and has no moving parts to get damaged.

Flash drives currently have a seek time of around 50 microseconds, which is about 100 times faster than disk drives. The transfer rate of current flash drives depends on the bus interface is it connected to. Flash drives connected by fast internal buses are comparable to those of disk drives; however, external USB flash drives are slower than disk drives.

Flash memory wears out. Each byte can be rewritten only a certain number of times; after the maximum is hit, the flash drive will fail. Currently this maximum is in the millions, which is reasonably high for most database applications. High-end drives employ “wear-leveling” techniques, which automatically move frequently-written bytes to less-written locations; this technique allows the drive to operate until all of the bytes on the drive reach their rewrite limit, instead of just the first one.

A flash drive presents a sector-based interface to the operating system, which means that a flash drive looks like a disk drive to the operating system. It is possible to employ RAID techniques with flash drives, although striping is less important because the seek time of a flash drive is so low.

The main impediment to flash drive adoption is its price. Prices are currently about 100 times the price of a comparable disk drive. Although the price of both flash and disk technology will continue to decrease, eventually flash drives will be cheap enough to be treated as mainstream. At that point, disk drives may be relegated to archival storage and the storage of extremely large databases.

Flash memory can also be used to enhance a disk drive by serving as a persistent front end. If the database fits entirely in the flash memory, then the disk drive will never get used. But as the database gets larger, the less frequently-used sectors will migrate to disk.

As far as a database system is concerned, a flash drive has the same properties as a disk drive: it is persistent, slow, and is accessed in sectors. (It just happens to be less slow than a disk drive.) Consequently, we shall conform to current terminology and refer to persistent memory as “the disk” throughout the rest of this book.

12.2 The Block-level Interface to the Disk

Disks may have different hardware characteristics – for example, they need not have the same sector size, and their sectors may be addressed in different ways. The operating system is responsible for hiding these (and other) details, providing its applications with a simple interface for accessing disks.

The notion of a *block* is central to this interface.

> *A block is similar to a sector, except that its size is determined by the OS. Each block has the same fixed size for all disks.*

The OS maintains the mapping between blocks and sectors. The OS also assigns a *block number* to each block of a disk; given a block number, the OS determines the actual sector addresses.

The contents of a block cannot be accessed directly from the disk. Instead, the sectors comprising the block must first be read into a memory *page* and accessed from there.

> *A page is a block-sized area of main memory.*

If a client wants to modify the contents of a block, the client reads the block into a page, modifies the bytes in the page, and then writes the page back to the block on disk.

An OS typically provides several methods to access disk blocks. Four such methods are:

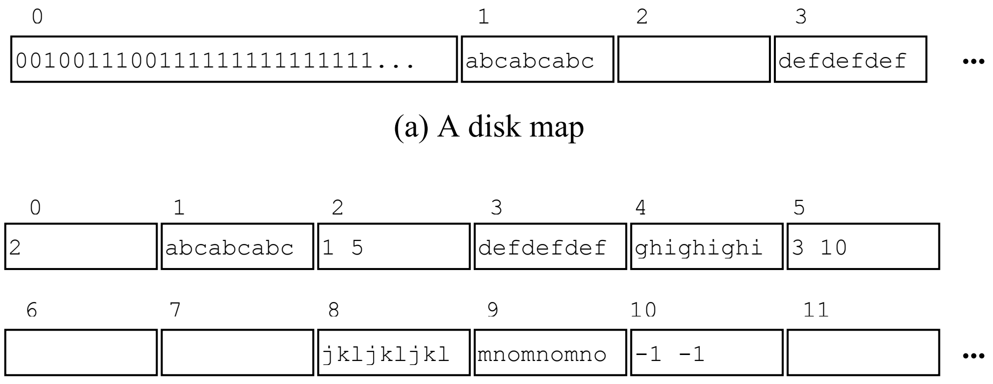
- *readblock(n, p)* reads the bytes at block *n* of the disk into page *p* of memory.
- *writeblock(n, p)* writes the bytes in page *p* of memory to block *n* of the disk.
- *allocate(k, n)* finds *k* contiguous unused blocks on disk, marks them as used, and returns the block number of the first one. The new blocks should be located as close to block *n* as possible.
- *deallocate(k, n)* marks the *k* contiguous blocks starting with block *n* as unused.

The OS keeps track of which blocks on disk are available for allocation and which are not. There are two basic strategies that it can adopt: a *disk map*, or a *free list*.

A disk map is a sequence of bits, one bit for each block on the disk. A bit value of 1 means the block is free, and a 0 means that the block is already allocated. The disk map is stored on disk, usually in the first several blocks. The OS can deallocate block *n* by simply changing bit *n* of the disk map to 1. It can allocate *k* contiguous blocks by searching the disk map for *k* bits in a row having the value 1, and then setting those bits to 0.

A free list is a chain of *chunks*, where a chunk is a contiguous sequence of unallocated blocks. The first block of each chunk stores two values: the length of the chunk, and the block number of the next chunk on the chain[†]. The first block of the disk contains a pointer to the first chunk on the chain. When the OS is asked to allocate k contiguous blocks, it searches the free list for a sufficiently large chunk. It then has the choice of removing the entire chunk from the free list and allocating it, or of splitting off a piece of length k and allocating only those blocks. When asked to deallocate a chunk of blocks, the OS simply inserts it into the free list.

Figure 12-6 illustrates these two techniques for a disk that has blocks 0, 1, 3, 4, 8, and 9 allocated. Figure 12-6(a) shows the disk map stored in block 0 of the disk; a bit value of 0 indicates an allocated block. Figure 12-6(b) shows the corresponding free list. Block 0 contains the value 2, meaning that the first chunk of the free list begins at block 2. Block 2 contains the two values 1 and 5, indicating that the chunk contains 1 block and that the next chunk begins at block 5. Similarly, the contents of block 5 indicate that its chunk is 3 blocks long, and the next chunk is at block 10. The value of -1 in block 10 indicates that it is the last chunk, which contains all remaining blocks.



(a) A disk map

(b) A free list

Figure 12-6: Two ways to keep track of free blocks

The free list technique requires minimal extra space; all we need is to store an integer in block 0 to point to the first block in the list. On the other hand, the disk map technique

[†] Since the block is unallocated, its contents can be used by the OS for any purpose whatsoever. In this case, it is a simple matter to use the first 8 bytes of the block to store these two integers.

requires space to hold the map. Figure 12-6(a) assumes that the map can fit into a single block. In general, however, several blocks may be required; see Exercise 12.6. The advantage of a disk map is that it gives the OS a better picture of where the "holes" in the disk are. For example, disk maps are often the strategy of choice if the OS needs to support the allocation of multiple blocks at a time.

12.3 The File-Level Interface to the Disk

The OS provides another, higher-level interface to the disk, called the *file system*. A client views a file as a named sequence of bytes. There is no notion of block at this level. Instead, a client can read (or write) any number of bytes starting at any position in the file.

The Java class *RandomAccessFile* provides a typical API to the file system. Each *RandomAccessFile* object holds a *file pointer* that indicates the byte at which the next read or write operation will occur. This file pointer can be set explicitly by a call to *seek*. A call to *readInt* (or *writeInt*) will also move the file pointer past the integer it read (or wrote).

An example is the code fragment in Figure 12-7, which increments the integer stored at bytes 7992-7995 of the file *junk*. The call to *readInt* reads the integer at byte 7992 and moves the file pointer to byte 7996. The subsequent call to *seek* sets the file pointer back to byte 7992, so that the integer at that location can be overwritten.

```
RandomAccessFile f = new RandomAccessFile("junk", "rws");
f.seek(7992);
int n = f.readInt();
f.seek(7992);
f.writeInt(n+1);
f.close();
```

Figure 12-7: Using the file-system interface to the disk

Note that the calls to *readInt* and *writeInt* act as if the disk were being accessed directly, hiding the fact that disk blocks must be accessed through pages. An OS typically reserves memory for several of these pages, which are called *I/O buffers*. When a file is opened, the OS assigns a page from this pool for use by the file, unbeknownst to the client.

By hiding its use of I/O buffers from the user, the file system obscures its interaction with the disk. For example, it is not obvious from Figure 12-7 that the method *writeInt* requires more disk accesses than *readInt*. A call to *readInt* requires the OS to read the corresponding block into an I/O buffer, and then return the value at the appropriate location. However, a call to *writeInt* requires that the OS read the block into a buffer, modify the location in the buffer, and then write the buffer back to the disk.

A file can be thought of as a sequence of blocks. For example, if blocks are 4096 bytes long (i.e. 4K bytes), then byte 7992 is in block 1 of the file (i.e., its second block). Block references like “block 1 of the file” are called *logical* block references, because they tell us where the block is with respect to the file, but not where the block is on disk.

Given a particular file location, the *seek* method determines the actual disk block that holds that location.

The seek method performs two conversions:

- *It converts the specified byte position to a logical block reference.*
- *It converts the logical block reference to a physical block reference.*

The first conversion is easy – the logical block number is just the byte position divided by the block size. For example, assuming 4K-byte blocks, byte 7992 is in block 1 because 7992/4096 = 1 (integer division).

The second conversion is harder, and depends on how a file system is implemented. We shall consider three file implementation strategies: *contiguous allocation*, *extent-based allocation*, and *indexed allocation*.

Continuous allocation

Contiguous allocation is the simplest strategy, which stores each file as a sequence of contiguous blocks. To implement contiguous allocation, the OS just needs to keep track of the length of each file and the location of its first block. The OS stores this information in its file system directory. Figure 12-8 depicts the directory for a file system containing two files: a 48-block long file named *junk* that begins at block 32, and a 16-block long file named *temp* that begins at block 80.

Name	1^{st} Block	Length
junk	32	48
temp	80	16

Figure 12-8: A file system directory for contiguous allocation

Mapping logical to physical block references is easy – if the file begins at disk block b, then block N of the file is in disk block b+N. There are two problems with this simple implementation. The first problem is that there is no way to extend a file if there is another file immediately following it. The file *junk* in Figure 12-8 is an example of such a file. Thus a file must be created with the maximum number of blocks it might need, which leads to wasted space when the file is not yet full. The second problem is that as the disk gets full, it may have lots of small-sized chunks of unallocated blocks, but no large chunks. Thus it may not be possible to create a large file, even though the disk contains plenty of free space. This first problem is known as *internal fragmentation*; the second problem is known as *external fragmentation.*

> *Internal fragmentation is the wasted space inside of a file.*
> *External fragmentation is the wasted space is outside of all the files.*

Extent-based allocation

The extent-based allocation strategy is a variation of contiguous allocation that reduces both internal and external fragmentation. Here, the OS stores a file a sequence of fixed-length *extents*, where each extent is a contiguous chunk of blocks. A file is extended one extent at a time. The OS implements this strategy by storing, for each file, a list of the first blocks of each extent in each file.

For example, suppose that the OS stores files in 8-block extents. Figure 12-9 depicts the file system directory for the two files *junk* and *temp*. These files have the same size as before, but now they are split into extents. The file *junk* has 6 extents, and the file *temp* has 2 extents.

Name	Extents
junk	32, 480, 696, 72, 528, 336
temp	64, 8

Figure 12-9: A file system directory for extent-based allocation

To find the disk block that holds block N of the file, the OS searches the extent list for that file to determine which extent covers that block, and then performs the arithmetic. Consider the following example.

> Suppose that the operating system has the file directory of Figure 12-9.
>
> To find block 21 of the file *junk*:
>
> The extent containing block 21 is `21 / 8 = 2` (integer division).
> The first logical block of that extent is `2 * 8 = 16`
> So block 21 is in block `21-16 = 5` of that extent.
> The extent begins at disk block `L[2] = 696`.
> Block 21 is thus at disk block `696 + 5 = 701`.

Note how extent-based allocation reduces internal fragmentation, because a file can waste no more than an extent's worth of space. External fragmentation is similarly reduced, because no more than an extent's worth of space need be available.

Indexed allocation

Indexed allocation takes a different approach – it doesn't even try to allocate files in contiguous chunks. Instead, each block of the file is allocated individually (in 1-block-long extents, if you will). The OS implements this strategy by allocating a special *index block* with each file, which keeps track of the disk blocks allocated to that file. That is, an index block *ib* can be thought of as an array of integers, where the value of ib[N] is the disk block that holds logical block N of the file. Calculating the physical disk block for any logical block is thus trivial – you just look it up in the index block.

Figure 12-10(a) depicts the file system directory for the two files *junk* and *temp*. The index block for junk is block 34. Figure 12-10(b) gives the first few integers in that block. From this figure, it is easy to see that block 1 of file *junk* is at block 103 of the disk.

Name	Index Block
junk	34
temp	439

(a) The directory table

```
Block 34:
    32  103  16  17  98 ...
```

(b) The contents of index block 34

Figure 12-10: A file system directory for indexed allocation

This approach has the advantage that blocks are allocated one at a time, so there is the least possible amount of fragmentation. Its main problem is that files will have a maximum size, because they can have only as many blocks as there are values in an index block. The UNIX file system addresses this problem by supporting multiple levels of index block, thereby allowing the maximum file size to be very large. See Exercises 12.12 and 12.13.

In each of these three implementation strategies, the OS needs to save the file system directory on the disk. The *seek* method accesses these directory blocks when it converts logical block references to physical block references. We can think of these disk accesses as "overhead" imposed by the file system. Operating systems try to minimize this overhead, but they cannot eliminate it. Consequently, every *seek* operation may cause one or more disk accesses to occur, somewhat unpredictably.

12.4 The Database System and the OS

The OS provides two levels of support for disk access: block-level support and file-level support. Which level should the implementers of a database system choose?

Choosing to use block-level support has the advantage of giving the database system complete control over which disk blocks are used for what purposes. For example, frequently-used blocks can be stored in the middle of the disk, where the seek time will be less. Similarly, blocks that tend to be accessed together can be stored near each other. Another advantage is that the database system is not constrained by OS limitations on files, allowing the database system to support tables that are larger than the OS limit, or that span multiple disk drives.

On the other hand, the use of the block-level interface has several disadvantages: Such a strategy is complex to implement; it requires that the disk be formatted and mounted as a *raw disk* – that is, a disk whose blocks are not part of the file system; and it requires that the database administrator have a lot of knowledge about block access patterns in order to fine-tune the system.

The other extreme is for the database system to use the OS file system as much as possible. For example, every table could be stored in a separate file, and the database system would access records by using file-level operations. This strategy is much easier to implement, and it allows the OS to hide the actual disk accesses from the database system. This situation is unacceptable for two reasons:

- The database system needs to know where the block boundaries are, so that it can organize and retrieve data efficiently.
- The database system needs to manage its own pages, because the OS way of managing I/O buffers is inappropriate for database queries.

We shall encounter these issues in later chapters.

A compromise strategy is for the database system to store all of its data in one or more OS files, but to treat the files as if they were raw disks. That is, the database system accesses its "disk" using logical file blocks. The OS is responsible for mapping each logical block reference to its corresponding physical block, via the *seek* method. Because the *seek* method may occasionally incur disk accesses, the database system will not be in complete control of the disk. However, the additional blocks accessed by the OS are usually insignificant compared with the large number of blocks accessed by the database system. Thus the database system is able to have the high-level interface to the OS, while maintaining significant control over when the disk gets accessed.

This compromise strategy is used in many database systems, such as MS Access (which keeps everything in a single *.mdb* file) and Oracle (which uses multiple OS files). SimpleDB also uses this compromise approach, similarly to Oracle.

12.5 The SimpleDB File Manager

As we have seen, an operating system provides important services that the database system needs, such as low-level disk handling and a file system. The portion of the database system that interacts with the OS is called the *file manager*.

> *The file manager is the database component*
> *responsible for interacting with the operating system.*

In this section we examine the file manager of the SimpleDB database system. Section 12.5.1 covers the file manager's API. We shall see how the file manager allows its clients to access any block of any file, reading a block to a memory page and writing the page back to its block in the file. The file manager also allows clients to place a value at an arbitrary position of a page, and to retrieve the value stored at that position. Section 12.5.2 then shows how the file manager can be implemented in Java.

12.5.1 Using the file manager

A SimpleDB database is stored in several files. There is a file for each table and each index, as well as a log file and several catalog files. The SimpleDB file manager provides block-level access to these files, via the package *simpledb.file*. This package exposes three classes: *Block*, *Page*, and *FileMgr*. Their API appears in Figure 12-11.

Block

```
    public Block(String filename, int blknum);
    public String filename();
    public int    number();
```

Page

```
    public Page();
    public int    getInt(int offset);
    public void   setInt(int offset, int val);
    public String getString(int offset);
    public void   setString(int offset, String val);

    public void   read(Block blk);
    public void   write(Block blk);
    public Block  append(String filename);
```

FileMgr

```
    public FileMgr(String dirname);
    public boolean isNew();
    public int     size(String filename);
```

Figure 12-11: The API for the SimpleDB file manager

A *Block* object identifies a specific block, by giving its file name and logical block number. For example, the statement

```
    Block blk = new Block("student.tbl", 23)
```

creates a new *Block* object that refers to block 23 of the file *student.tbl*. Its accessor methods *filename* and *number* extract the file name and block number from the block.

A *Page* object holds the contents of a disk block, and corresponds to an OS I/O buffer. The methods *setInt* and *setString* place a value at the specified offset of the page, and the methods *getInt* and *getString* retrieve a previously-stored value from the specified offset. Although integers and strings are currently the only supported values, methods for other types could be added; see Exercise 12.17. A client can store a value at any offset of the page, but is responsible for knowing what values have been stored where. An attempt to get a value from the wrong offset will have unpredictable results.

The methods *read*, *write*, and *append* access the disk. The *read* method reads the contents of the specified block into the page; method *write* does the reverse. The *append* method allocates a new block at the end of the specified file, initializes its contents to the contents of the specified page, and returns the logical reference to the newly-allocated block.

The *FileMgr* class handles the actual interaction with the OS file system. Its constructor takes a string as argument, which denotes the name of the folder in the user's home directory where the database files live. If there is no such folder, then a folder is created for a new database. The method *isNew* returns *true* in this case, and *false* otherwise. This method is needed for the proper initialization of the server.

The method *size* returns the number of blocks in the specified file, which allows clients to move directly to the end of the file.

Each instance of the system has one *FileMgr* object, which is created during system startup. The class *server.SimpleDB* creates the object; it has a static method *fileMgr* that returns the created object.

Figure 12-12 contains a code fragment that illustrates the use of these methods. This code has three sections. The first section initializes the SimpleDB system so that it uses the database in the *studentdb* directory, obtains the file manager, determines the last block of the file *junk*, and creates a *Block* object that refers to it. The second section increments the integer at offset 3992 of this block, analogous to the code of Figure 12-7. The third section stores the string "hello" at offset 20 of another page, and then appends it to a new block of the file. It then reads that block into a third page, extracts the value "hello" into variable *s*, and prints the number of the new block together with *s*.

```
SimpleDB.init("studentdb");
FileMgr fm = SimpleDB.fileMgr();
int filesize = fm.size("junk");
Block blk = new Block("junk", filesize-1);

Page p1 = new Page();
p1.read(blk);
int n = p1.getInt(3992);
p1.setInt(3992, n+1);
p1.write(blk);

Page p2 = new Page();
p2.setString(20, "hello");
blk = p2.append("junk");
Page p3 = new Page();
p3.read(blk);
String s = p3.getString(20);
System.out.println("Block " + blk.number() + " contains " + s);
```

Figure 12-12: Using the SimpleDB file manager

12.5.2 Implementing the file manager

We now consider how the three file manager classes get implemented.

The class *Block*

The code for class *Block* appears in Figure 12-13. In addition to straightforward implementations of the methods *filename* and *number*, the class also implements *equals, hashCode,* and *toString*.

```
public class Block {
   private String filename;
   private int blknum;

   public Block(String filename, int blknum) {
      this.filename = filename;
      this.blknum   = blknum;
   }

   public String filename() {
      return filename;
   }

   public int number() {
      return blknum;
   }

   public boolean equals(Object obj) {
      Block blk = (Block) obj;
      return filename.equals(blk.filename) && blknum == blk.blknum;
   }

   public String toString() {
      return "[file " + filename + ", block " + blknum + "]";
   }

   public int hashCode() {
      return toString().hashCode();
   }
}
```

Figure 12-13: The code for the SimpleDB class *Block*

The class *Page*

The code to implement class *Page* appears in Figure 12-14.

```
public class Page {
   public static final int BLOCK_SIZE = 400;
   public static final int INT_SIZE = Integer.SIZE / Byte.SIZE;
   public static final int STR_SIZE(int n) {
      float bytesPerChar =
                  Charset.defaultCharset().newEncoder().maxBytesPerChar();
      return INT_SIZE + (n * (int)bytesPerChar);
   }

   private ByteBuffer contents = ByteBuffer.allocateDirect(BLOCK_SIZE);
   private FileMgr    filemgr  = SimpleDB.fileMgr();

   public synchronized void read(Block blk) {
      filemgr.read(blk, contents);
   }

   public synchronized void write(Block blk) {
      filemgr.write(blk, contents);
   }

   public synchronized Block append(String filename) {
      return filemgr.append(filename, contents);
   }

   public synchronized int getInt(int offset) {
      contents.position(offset);
      return contents.getInt();
   }

   public synchronized void setInt(int offset, int val) {
      contents.position(offset);
      contents.putInt(val);
   }

   public synchronized String getString(int offset) {
      contents.position(offset);
      int len = contents.getInt();
      byte[] byteval = new byte[len];
      contents.get(byteval);
      return new String(byteval);
   }

   public synchronized void setString(int offset, String val) {
      contents.position(offset);
      byte[] byteval = val.getBytes();
      contents.putInt(byteval.length);
      contents.put(byteval);
   }
}
```

Figure 12-14: The code for the SimpleDB class *Page*

The class defines three public constants: BLOCK_SIZE, INT_SIZE, and STR_SIZE. The constant BLOCK_SIZE denotes the number of bytes in a block. The value shown, 400, is set artificially low so that we can easily create demo databases having a lot of blocks. In a commercial database system, this value would be set to the block size defined by the operating system; a typical block size is 4K bytes.

The constants INT_SIZE and STR_SIZE denote the number of bytes required to store integer and string values. Integer values have a standard representation in Java, which is always 4 bytes long. However, there is no standard Java string representation. SimpleDB represents a string in two parts: The second part is the byte representation of the characters in the string, and the first part is the length of that representation. So for example if we assume that each character is represented using one byte (as in the character set for English), then the string "joe" will be represented by the 4-byte integer representation of 3, followed by the byte representations of 'j', 'o', and 'e'. In general, STR_SIZE(n) returns the value 4+(n*k), where k is the maximum number of bytes required to represent a character in the default character set.

The actual contents of a page are stored as a Java *ByteBuffer* object. Instead of a public constructor, *ByteBuffer* contains the two static factory methods *allocate* and *allocateDirect*. The code for *Page* uses the *allocateDirect* method, which tells the compiler to use one of the operating system's I/O buffers to hold the byte array.

A *ByteBuffer* object keeps track of its current position, which can be set by the method *position.* The *get/put* methods in *Page* simply set the position to the specified offset, and then call the corresponding *ByteBuffer* method to access the byte array at that position. The *getString* method first reads an integer, which denotes the number of bytes in the string; it then reads that many bytes and constructs the string from them. The *putString* method performs the reverse actions.

All of the methods in class *Page* are *synchronized*, which means that only one thread can be executing them at a time. Synchronization is needed to maintain consistency when methods share an updateable object, such as the variable *contents* in Figure 12-14. For example, the following scenario could occur if *getInt* were not synchronized: Suppose that two JDBC clients, each running in their own thread, are trying to read different integers on the same page. Thread A runs first. It starts to execute *getInt*, but gets interrupted right after the first line is executed; that is, it has positioned the byte buffer but has not yet read from it. Thread B runs next, and executes *getInt* to completion. When thread A resumes, the bytebuffer's position will have changed, but the thread will not notice it; thus, it will incorrectly read from the wrong position.

The class *FileMgr*

The code for class *FileMgr* appears in Figure 12-15. Its primary job is to implement the methods *read, write*, and *append* that are defined in *Page*. Its methods *read* and *write* seek to the appropriate position in the specified file, and read (or write) the block's contents to the specified byte buffer. The *append* method seeks to the end of the file and then writes the specified byte buffer to it; the OS will automatically extend the file.

```
public class FileMgr {
   private File dbDirectory;
   private boolean isNew;
   private Map<String,FileChannel> openFiles =
                              new HashMap<String,FileChannel>();

   public FileMgr(String dbname) {
      String homedir = System.getProperty("user.home");
      dbDirectory = new File(homedir, dbname);
      isNew = !dbDirectory.exists();

      // create the directory if the database is new
      if (isNew && !dbDirectory.mkdir())
         throw new RuntimeException("cannot create " + dbname);

      // remove any leftover temporary tables
      for (String filename : dbDirectory.list())
         if (filename.startsWith("temp"))
            new File(dbDirectory, filename).delete();
   }

   synchronized void read(Block blk, ByteBuffer bb) {
      try {
         bb.clear();
         FileChannel fc = getFile(blk.fileName());
         fc.read(bb, blk.number() * bb.capacity());
      }
      catch (IOException e) {
         throw new RuntimeException("cannot read block " + blk);
      }
   }

   synchronized void write(Block blk, ByteBuffer bb) {
      try {
         bb.rewind();
         FileChannel fc = getFile(blk.fileName());
         fc.write(bb, blk.number() * bb.capacity());
      }
      catch (IOException e) {
         throw new RuntimeException("cannot write block" + blk);
      }
   }

   synchronized Block append(String filename, ByteBuffer bb) {
      try {
         FileChannel fc = getFile(filename);
         int newblknum = size(filename);
         Block blk = new Block(filename, newblknum);
         write(blk, bb);
         return blk;
      }
      catch (IOException e) {
         throw new RuntimeException("cannot access " + filename);
      }
   }
   public synchronized int size(String filename) {
      try {
         FileChannel fc = getFile(filename);
         return (int)(fc.size() / Page.BLOCK_SIZE);
      }
      catch (IOException e) {
         throw new RuntimeException("cannot access " + filename);
      }
   }

   public boolean isNew() {
      return isNew;
   }

   private synchronized FileChannel getFile(String filename)
                                    throws IOException {
      FileChannel fc = openFiles.get(filename);
      if (fc == null) {
         File dbTable = new File(dbDirectory, filename);
         RandomAccessFile f = new RandomAccessFile(dbTable, "rws");
         fc = f.getChannel();
         openFiles.put(filename, fc);
      }
      return fc;
   }
}
```

Figure 12-15: The code for the SimpleDB class *FileMgr*

Note how the file manager always reads or writes a block-sized number of bytes from a file, and always at a block boundary. In doing so, the file manager is ensuring that each call to *read*, *write*, or *append* will entail exactly one disk access.

In Java, a file must be opened before it can be used. The SimpleDB file manager opens a file by creating a new *RandomAccessFile* object for it and saving its file channel. The method *getFile* manages these *FileChannel* objects. Note that the files are opened in "rws" mode. The "rw" portion specifies that the file is open for reading and writing. The "s" portion specifies that the OS should not delay disk I/O in order to optimize disk performance; instead, every *write* operation must be written immediately to the disk. This feature ensures that the database system always knows exactly what the disk contains, which will be especially important when we consider recovery management in Chapter 14.

There is only one *FileMgr* object in SimpleDB, which is created by the method *SimpleDB.init* in package *simpledb.server*. The *FileMgr* constructor determines if the specified database folder exists, and creates it if necessary. The constructor also removes any temporary files that might have been created by the materialized operators of Chapter 22.

12.5.3 Unit testing the file manager

Some of the end-of-chapter exercises ask you to modify the SimpleDB *file* package. How do you test your modified code? A bad testing strategy is to just plug the modified code into the server, and see if the JDBC clients run on it. Not only is debugging difficult and tedious using this strategy (for example, error messages frequently do not correspond to the bug), but there is no way to know if the JDBC clients are fully testing the modified code.

A better testing strategy is called *unit testing*. In this strategy, you create a *test driver* for each modified class (or package); the driver should test the code on enough cases that you are convinced that the code works. For example, the code fragment of Figure 12-12 could be part of a test driver for the *simpledb.file* package.

In general, a test driver should be in the same package as the code it is testing, so that it can call package-private methods. Consequently, the code must be run using the complete name of the class. For example, the driver class *FileTest* might be called as follows:

```
> java  simpledb.file.FileTest
```

12.6 Chapter Summary

- A *disk drive* contains one or more rotating *platters*. A platter has concentric *tracks*, and each track consists of *sectors*. The size of a sector is determined by the disk manufacturer; a common sector size is 512 bytes.

- Each platter has its own read/write head. These heads do not move independently; instead, they are all connected to a single *actuator*, which moves them simultaneously to the same track on each platter.

- A disk drive executes a disk access in three stages:
    - The actuator moves the disk head to the specified track. This time is called the *seek time*.
    - The drive waits for the platter to rotate until the first desired byte is beneath the disk head. This time is called the *rotational delay*.
    - The bytes rotating under the disk head are read (or written). This time is called the *transfer time*.

- Disk drives are slow, because their activity is mechanical. Access times can be improved by using disk caches, cylinders, and disk striping. A disk cache allows the disk to pre-fetch sectors by reading an entire track to be read at a time. A cylinder consists of the tracks on each platter having the same track number. Blocks on the same cylinder can be accessed with no additional seek time. Disk striping distributes the contents of a virtual disk among several small disks. Speedup occurs because the small disks can operate simultaneously.

- RAID techniques can be used to improve disk reliability. The basic RAID levels are:
  - RAID-0 is striping, with no additional reliability. If a disk fails, the entire database is effectively ruined.
  - RAID-1 adds mirroring to the striped disks. Each disk has an identical mirror disk. If a disk fails, its mirror can be used to reconstruct it.
  - RAID-4 uses striping with an additional disk to hold redundant parity information. If a disk fails, its contents can be reconstructed by combining the information on the other disks with the parity disk.

- The RAID techniques require a *controller* to hide the existence of the multiple disks from the operating system and provide the illusion of a single, virtual disk. The controller maps each virtual read/write operation to one or more operations on the underlying disks.

- Disk technology is being challenged by flash memory. Flash memory is persistent, but faster than disk because it is completely electronic. However, since flash is still significantly slower than RAM, the operating system treats a flash drive the same as a disk drive.

- The operating system hides the physical details of disk and flash drives from its clients by providing a block-based interface to them. A *block* is similar to a sector, except that its size is OS-defined. A client accesses the contents of a device by block number. The OS keeps track of which blocks on disk are available for allocation, by using either a *disk map* or a *free list*.

- A *page* is a block-sized area of memory. A client modifies a block by reading its contents into a page, modifying the page, and then writing the page back to the block.

- The OS also provides a file-level interface to the disk. A client views a file as a named sequence of bytes.

- An OS can implement files using *contiguous allocation*, *extent-based allocation*, or *indexed allocation*. Contiguous allocation stores each file as a sequence of contiguous blocks. Extent-based allocation stores a file a sequence of *extents*, where each extent is a contiguous chunk of blocks. Indexed allocation allocates each block of the file individually. A special *index block* is kept with each file, to keep track of the disk blocks allocated to that file.

- A database system can choose to use either the block-level or the file-level interface to the disk. A good compromise is to store the data in files, but to access the files at the block level.

12.7 Suggested Reading

A basic coverage of disk drives and RAID strategies can be found online at *www.pcguide.com/ref/hdd*. The article [Chen et al 1994] provides a detailed survey of the various RAID strategies and their performance characteristics. A good book that discusses UNIX-based file systems is [von Hagen 2002], and one that discusses Windows NTFS is [Nagar 1997]. Brief overviews of various OS file system implementations can be found in many operating systems textbooks, such as [Silberschatz et al. 2004].

A property of flash memory is that overwriting an existing value is significantly slower than writing a completely new value. Consequently, there has been a lot of research aimed at flash-based file systems that are do not overwrite values. Such file systems store updates in a log, similar to the log of Chapter 13. The articles [Wu and Kuo 2006] and [Lee and Moon 2007] examine these issues.

12.8 Exercises

CONCEPTUAL EXERCISES

12.1 Consider a single-platter disk containing 50,000 tracks and spinning at 7200rpm. Each track holds 500 sectors, and each sector contains 512 bytes.
a) What is the capacity of the platter?
b) What is the average rotational delay?
c) What is the maximum transfer rate?

12.2 Consider an 80GB disk drive spinning at 7200 rpm with a transfer rate of 100MB/sec. Assume that each track contains the same number of bytes.
a) How many bytes does each track contain? How many tracks does the disk contain?
b) If the disk were spinning at 10,000 rpm, what would the transfer rate be?

12.3 Suppose that you have 10 20GB disk drives, each of which has 500 sectors per track. Suppose that you want to create a virtual 200GB drive by striping the small disks, with the size of each stripe being an entire track instead of just a single sector.
a) Suppose that the controller receives a request for virtual sector M. Give the formula that computes the corresponding actual drive and sector number.
b) Give a reason why track-sized stripes might be more efficient than sector-sized stripes.
c) Give a reason why track-sized stripes might be less efficient than sector-sized stripes.

12.4 All of the failure-recovery procedures discussed in this chapter require the system to be shut down while the failed disk is replaced. Many systems cannot tolerate downtime of any amount, and yet they also don't want to lose data.
a) Consider the basic mirroring strategy. Give an algorithm for restoring a failed mirror without any downtime. Does your algorithm increase the risk of a second disk failure? What should be done to reduce this risk?
b) Modify the parity strategy to similarly eliminate downtime. How do you deal with the risk of a second disk failure?

12.5 One consequence of the RAID-4 parity strategy is that the parity disk gets accessed for every disk write operation. One suggested improvement is to omit the parity disk, and instead "stripe" the data disks with the parity information. For example, sectors 0, N,

2N, etc. of disk 0 will contain parity information, as will sectors 1, N+1, 2N+1, etc. of disk 1, and so on. (This improvement is called *RAID-5*.)
a) Suppose a disk fails. Explain how it will get recovered.
b) Show that with this improvement, disk reads and writes still require the same number of disk accesses as RAID-4.
c) Explain why this improvement nevertheless leads to more efficient disk accesses.

12.6 Consider Figure 12-5, and suppose that one of the striped disks fails. Show how to reconstruct its contents using the parity disk.

12.7 Figure 12-16 depicts a RAID-4 system in which one of the disks has failed. Use the parity disk to reconstruct its values.

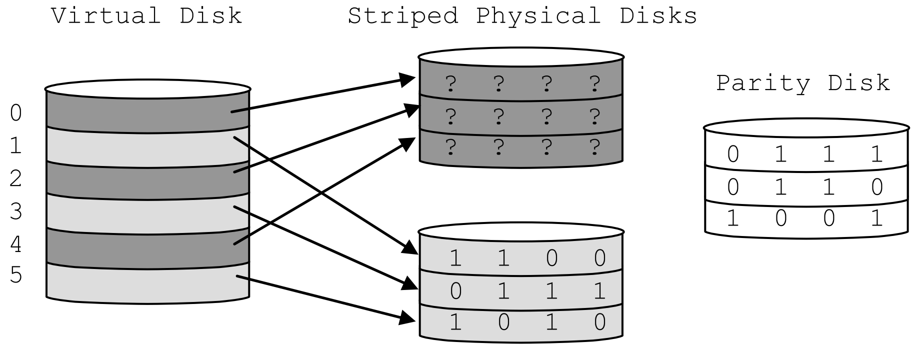


Figure 12-16: A failed physical disk in a RAID-4 system

12.8 Consider a 1GB database, stored in a file whose block size is 4K bytes.
a) How many blocks will the file contain?
b) Suppose that the database system uses a disk map to manage its free blocks. How many additional blocks are needed to hold the disk map?

12.9 Consider Figure 12-6. Draw a picture of the disk map and the free list after the following operations have been executed:

```
allocate(1,4); allocate(4,10); allocate(5,12);
```

12.10 The free list allocation strategy can wind up with two contiguous chunks on the free list.
(a) Explain how to modify the free-list technique so that contiguous chunks can be merged.
(b) Explain why merging unallocated chunks is a good idea when files are allocated contiguously.
(c) Explain why merging is not important for extent-based or indexed file allocation.

12.11 Suppose that the OS uses extent-based file allocation using extents of size 12, and suppose that the extent list for a file is [240, 132, 60, 252, 12, 24].
(a) What is the size of the file?
(b) Calculate the physical disk block of logical blocks 2, 12, 23, 34, and 55 of the file.

12.12 Consider a file implementation that uses indexed file allocation. Assuming that the block size is 4K bytes, what is the size of the largest possible file?

12.13 In UNIX, the directory entry for a file points to a block called an *inode*. In one implementation of an inode, the beginning of the block holds various header information, and its last 60 bytes contains 15 integers. The first 12 of these integers are the physical locations of the first 12 data blocks in the file. The next two integers are the location of two *index blocks*, and the last integer is the location of a *double-index block*. An index block consists entirely of block numbers of the next data blocks in the file; a double-index block consists entirely of block numbers of index blocks (whose contents point to data blocks).
a) Assuming again that the block size is 4K bytes, how many data blocks does an index block refer to?
b) Ignoring the double-index block, what is the largest possible UNIX file size?
c) How many data blocks does a double-index block refer to?
d) What is the largest possible UNIX file size?
e) How many block accesses are required to read the last data block of a 1GB file?
f) Give an algorithm to implement the *seek* function for a UNIX file.

12.14 The movie and song title "On a clear day you can see forever" is occasionally misquoted as "On a clear disk you can seek forever". Comment on the cleverness of the pun.

PROGRAMMING EXERCISES

12.15 A database system often contains diagnostic routines.
a) Modify the class *FileMgr* so that it keeps useful statistics, such as the number of blocks read/written. Add new method(s) to the class that will return these statistics.
b) Modify the methods *commit* and *rollback* of the class *RemoteConnectionImpl* (in the *remote* package) so that they print these statistics. The result will be that the server prints the statistics for each SQL statement it executes. Note that your code can obtain the *FileMgr* object by calling the static method *SimpleDB.fileMgr* (in the *server* package).

12.16 The methods *setInt* and *setString* of class *Page* do not check that the new value fits in the page.
a) Modify the code to perform the checks. What should you do if the check fails?
b) Give a reason why it is reasonable to not perform the checks.

12.17 The class *Page* has methods to get/set only integers and strings. Modify the class to handle other types, such as short integers, booleans, byte arrays, and dates.

12.18 The class *Page* implements a string by prepending its length to the string's characters. Another way to implement a string is to append a delimiter character after the string's characters. A reasonable delimiter character in Java is '\0'. Modify the class accordingly.

13
MEMORY MANAGEMENT
and the packages simpledb.log *and* simpledb.buffer

In this chapter we shall study two database components: the *log manager* and the *buffer manger*. Each of these components is responsible for a particular set of files: The log manager is responsible for the log file, and the buffer manager is responsible for the users' data files.

The key problem faced by these two components is how to efficiently manage the reading and writing of disk blocks with main memory. The contents of a database is typically much larger than main memory, and so the components may need to shuttle blocks in and out of memory. We shall examine the various memory-management algorithms used by the log manager and buffer manager. It turns out that the two components use very different algorithms, because they are used in very different ways. The log manager has a simple, optimal algorithm for managing the log file, whereas the buffer manager algorithms can only be best-guess approximations.

13.1 Two Principles of Database Memory Management

As we have seen, database data is stored in disk blocks. In order to access a desired block, the database system must read it into a memory page. Database systems follow two important principles when they move data between the disk and memory.

> *There are two fundamental principles of memory management:*
> 1. *Minimize disk accesses.*
> 2. *Don't rely on virtual memory.*

Principle 1: Minimize Disk Accesses

Consider an application that reads data from the disk, searches through the data, performs various computations, makes some changes, and writes the data back. How can we estimate the amount of time this will take? Recall that RAM operations are over 1,000 times faster than flash and 100,000 times faster than disk. This means that in most practical situations, the time it takes to read/write the block from disk is at least as large as the time it takes to process the block in RAM. Consequently, the single most important thing a database system can do is minimize block accesses.

One way to minimize block accesses is to avoid repeated accesses of a disk block. This kind of problem occurs in many areas of computing, and has a standard solution known as *caching*. For example, a CPU has a local hardware cache of previously-executed instructions; if the next instruction is in the cache, the CPU does not have to load it from RAM. For another example, a browser keeps a cache of previously-accessed web pages; if a user requests a page that happens to be in the cache (say, by hitting the browser's *Back* button), the browser can avoid retrieving it from the network.

A database system caches disk blocks in its memory pages. It keeps track of which pages contain the contents of which blocks; in this way, it may be possible to satisfy a client request by using an existing page, thereby avoiding a disk read. Similarly, a database system writes pages to disk only when necessary, in the hope that multiple changes to a page can be made via a single disk write.

The need to minimize disk accesses is so important that it pervades the entire working of the database system. For example, the retrieval algorithms used by a database system are chosen specifically because of the frugal way that they access the disk. And when an SQL query has several possible retrieval strategies, the planner will choose the strategy that it thinks will require the fewest number of disk accesses.

Principle 2: Don't Rely on Virtual Memory

Modern operating systems support *virtual memory*. The OS gives each process the illusion that it has a very large amount of memory in which to store its code and data. A process allocates objects arbitrarily in its virtual memory space; the OS maps each virtual page to an actual page of physical memory.

The virtual memory space supported by an OS is usually far larger than a computer's physical memory. Since not all virtual pages fit in physical memory, the OS must store some of them on disk. When a process accesses a virtual page not in memory, a *page swap* occurs: The OS chooses a physical page, writes the contents of that page to disk (if it had been modified), and reads the saved contents of the virtual page from disk to that page.

The most straightforward way for the database system to manage disk blocks is to give each block its own page. For example, it could keep an array of pages for each file, having one slot for each block of the file. These arrays would be huge, but they would fit in virtual memory. As the database system accessed these pages, the OS virtual memory mechanism would swap them between disk and physical memory, as needed. This is a simple, easily-implemented strategy. Unfortunately, it has a serious problem: The OS, not the database system, controls when pages get written to disk. Two issues are at stake.

The first issue is that the OS's page-swapping strategy can impair the database system's ability to recover after a system crash. The reason, as we shall see in Chapter 14, is that a modified page will have some associated log records, and these log records *must* be written to disk before the page. (Otherwise, the log records will not be available when the database is recovered after a system crash.) Since the OS does not know about the log, it may swap out a modified page without writing its log records, and thereby subvert the recovery mechanism.[†]

The second issue is that the OS has no idea which pages are currently in use and which pages the database system no longer cares about. The OS can make an educated guess, such as choosing to swap the page that was least recently accessed. But if the OS guesses incorrectly, it will swap out a page that will be needed again, causing two unnecessary disk accesses. A database system, on the other hand, has a much better idea of what pages are needed, and can make much more intelligent guesses.

Therefore, a database system must manage its own pages. It does so by allocating a relatively small number of pages that it knows will fit in physical memory; these pages are known as the database's *buffer pool*. The database system keeps track of which pages are available for swapping. When a block needs to be read into a page, the database system (and not the OS) chooses an available page from the buffer pool, writes its contents (and its log record) to disk if necessary, and only then reads in the specified block.

13.2 Managing Log Information

Whenever a user changes the database, the database system must keep track of that change, just in case it needs to be undone. The values describing a change are kept in a *log record*, which are stored in a *log file*. New log records are always appended to the end of the log.

The log manager is the portion of the database system
that is responsible for writing log records to the log file.

The log manager does not understand the contents of the log records – that responsibility belongs to the recovery manager (see Chapter 14). Instead, the log manager treats the log as just an ever-increasing sequence of log records.

In this section we examine how the log manager can manage memory as it writes log records to the log file. We begin by considering the algorithm of Figure 13-1, which is the most straightforward way to append a record to the log.

[†] Actually, there do exist operating systems that address this issue, but they are not commonplace.

1. Allocate a page in memory.
2. Read the last block of the log file into that page.

3a. If there is room, put the log record after the other records on the page, and write the page back to disk.

3b. If there is no room, then allocate a new, empty page, place the log record in that page, and append the page to a new block at the end of the log file.

Figure 13-1: A simple (but inefficient) algorithm for appending a new record to the log

This algorithm requires a disk read and a disk write for every appended log record. It is simple, but very inefficient.

Figure 13-2 illustrates the operation of the log manager halfway through step 3a of the algorithm. The log file contains three blocks that hold eight records, labeled r1 through r8. Log records can have varying sizes, which is why four records fit into block 0 but only three fit into block 1. Block 2 is not yet full, and contains just one record. The memory page contains the contents of block 2. In addition to record r8, a new log record (record r9) has just been placed in the page.

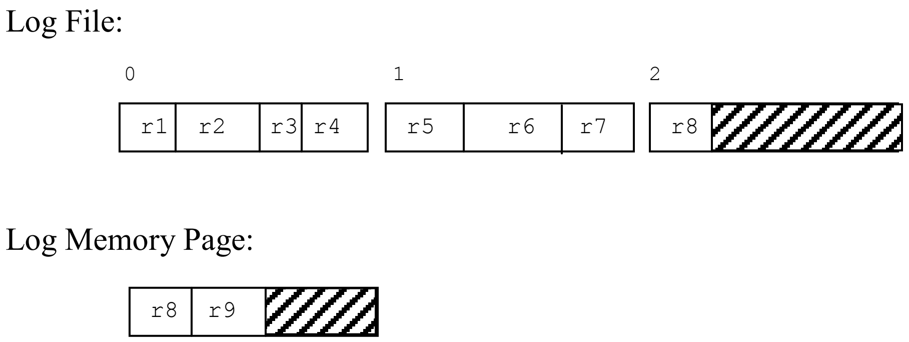


Figure 13-2: Adding a new log record r9

Suppose now that the log manager completes the algorithm by writing the page back to block 2 of the file. When the log manager is eventually asked to add another log record to the file, it will perform steps 1 and 2 of the algorithm and read block 2 into a page. But note that this disk read is completely unnecessary, because the existing log page already contains the contents of block 2! Consequently, steps 1 and 2 of the algorithm are unnecessary. The log manager just needs to permanently allocate a page to contain the contents of the last log block. As a result, all of the disk reads are eliminated.

It is also possible to reduce the disk writes. In the above algorithm, the log manager writes its page to disk every time a new record is added to the page. Looking at Figure 13-2, we can see that there is no need to write record *r9* immediately to the disk. Each new log record can be simply added to the page, as long as the page has room. When the page becomes full, the log manager can write the page to disk, clear its contents, and start anew. This new algorithm would result in exactly one disk write for each log block, which is clearly optimum.

Our new algorithm has one glitch: A log page may need to be written to the disk before it is full, due to circumstances beyond the control of the log manager. The issue is that the buffer manager cannot write a modified data page to disk until its associated log records are also be written to disk. If one of those log records happen to be in the log page but not yet on disk, then the log manager must write its page to disk, regardless of whether the page is full.

This discussion results in the algorithm of Figure 13-3.

```
1. Permanently allocate one memory page to hold the contents
   of the last block of the log file. Call this page P.

2. When a new log record is submitted:
   a) If there is no room in P, then:
          Write P to disk and clear its contents.
   b) Add the new log record to P.

3. When the database system requests that a particular log record be written to disk:
   a) Determine if that log record is in P.
   b) If so, then write P to disk.
```

Figure 13-3: The optimal log management algorithm

In other words, there are two reasons why a memory page gets written to disk:

- whenever a log record needs to be forced to disk;
- when the page is full.

Consequently, a memory page might get written to the same log block multiple times. But since these disk writes are absolutely necessary and cannot be avoided, we can conclude that the algorithm is optimal.

13.3 The SimpleDB Log Manager

This section examines the log manager of the SimpleDB database system. Section 13.3.1 covers the log manager's API, which includes methods for appending a log record to the log, and for reading through the records in the log file. Section 13.3.2 then shows how the log manager can be implemented in Java.

13.3.1 The API for the log manager

The SimpleDB log manager implementation is in the package *simpledb.log*. This package exposes the two classes *LogMgr* and *BasicLogRecord*; their API appears in Figure 13-4.

LogMgr

```
public LogMgr(String logfile);
public int   append(Object[] rec);
public void  flush(int lsn);
public Iterator<BasicLogRecord> iterator();
```

BasicLogRecord

```
public BasicLogRecord(Page pg, int pos);
public int    nextInt();
public String nextString();
```

Figure 13-4: The API for the SimpleDB log manager

The database system has one *LogMgr* object, which is created during system startup. The class *server.SimpleDB* creates the object by passing the name of the log file into the constructor; it has a static method *logMgr* that returns the created object.

The method *append* adds a record to the log, and returns an integer. As far as the log manager is concerned, a log record is just an arbitrarily-sized array of values. The only constraints are that each value must be an integer or a string, and the record must fit inside of a page. The return value from *append* identifies the new log record; this identifier is called its *log sequence number* (or *LSN*).

An LSN is a value that identifies a record on the log.

Appending a record to the log does not guarantee that the record will get written to disk; instead, the log manager chooses when to write log records to disk, as in the algorithm of Figure 13-3. A client can force a specific log record to disk by calling the method *flush*. The argument to *flush* is the LSN of the log record; the method ensures that the log record (and all previous log records) are written to disk.

A client can read the records in the log by calling the method *iterator*; this method returns a Java iterator for the log records. Each call to the iterator's *next* method will return a *BasicLogRecord* object for the next record in the log. A basic log record is just a sequence of bytes on a page; the caller is responsible for knowing how many values are in the record and what their types are. The only way to read the values in a log record is to call the methods *nextInt* and *nextString*. The method *nextString* assumes that the next value in the log record is a string and returns it, and similarly for the method *nextInt*.

The records returned by the *iterator* method are in reverse order, starting at the most recent record and moving backwards through the log file. The records are returned in this order because that is how the recovery manager wants to see them.

The code fragment of Figure 13-5(a) provides an example of how to use the log manager API. The first half of the code appends three records to the log file. Each log record consists of two strings: the first record is [“a”, “b”], the second [“c”, “d”], and the third [“e”, “f”]. The code then calls *flush*, passing it the LSN of the third log record; this call ensures that all three log records will be written to disk. The second half of the code iterates through the records in the log file, in reverse order, and prints their strings. The output appears in Figure 13-5(b).

```
SimpleDB.init("studentdb");
LogMgr logmgr = SimpleDB.logMgr();
int lsn1 = logmgr.append(new Object[]{"a", "b"});
int lsn2 = logmgr.append(new Object[]{"c", "d"});
int lsn3 = logmgr.append(new Object[]{"e", "f"});
logmgr.flush(lsn3);

Iterator<BasicLogRecord> iter = logmgr.iterator();
while (iter.hasNext()) {
      BasicLogRecord rec = iter.next();
      String v1 = rec.nextString();
      String v2 = rec.nextString();
      System.out.println("[" + v1 + ", " + v2 + "]");
}
```

(a) An example code fragment

```
[e, f]
[c, d]
[a, b]
```

(b) The output of the code fragment

Figure 13-5: Writing log records and reading them

13.3.2 Implementing the log manager

The class *LogMgr*

The code for *LogMgr* appears in Figure 13-6. Its constructor uses the provided filename as the log file. If the file is empty, the constructor creates the file by appending a new empty block to it. The constructor also allocates a single page (called *mypage*), and initializes it to contain the contents of the last log block in the file.

```
public class LogMgr implements Iterable<BasicLogRecord> {
   public static final int LAST_POS = 0;

   private String logfile;
   private Page mypage = new Page();
   private Block currentblk;
   private int currentpos;

   public LogMgr(String logfile) {
      this.logfile = logfile;
      int logsize = SimpleDB.fileMgr().size(logfile);
      if (logsize == 0)
         appendNewBlock();
      else {
         currentblk = new Block(logfile, logsize-1);
         mypage.read(currentblk);
         currentpos = getLastRecordPosition() + INT_SIZE;
      }
   }

   public void flush(int lsn) {
      if (lsn >= currentLSN())
         flush();
   }

   public Iterator<BasicLogRecord> iterator() {
      flush();
      return new LogIterator(currentblk);
   }

   public synchronized int append(Object[] rec) {
      int recsize = INT_SIZE;
      for (Object obj : rec)
         recsize += size(obj);
      if (currentpos + recsize >= BLOCK_SIZE){
         flush();
         appendNewBlock();
      }
      for (Object obj : rec)
         appendVal(obj);
      finalizeRecord();
      return currentLSN();
   }

   private void appendVal(Object val) {
      if (val instanceof String)
         mypage.setString(currentpos, (String)val);
      else
         mypage.setInt(currentpos, (Integer)val);
      currentpos += size(val);
   }

   private int size(Object val) {
      if (val instanceof String) {
         String sval = (String) val;
         return STR_SIZE(sval.length());
      }
      else
         return INT_SIZE;
   }

   private int currentLSN() {
      return currentblk.number();
   }

   private void flush() {
      mypage.write(currentblk);
   }

   private void appendNewBlock() {
      setLastRecordPosition(0);
      currentblk = mypage.append(logfile);
      currentpos = INT_SIZE;
   }

   private void finalizeRecord() {
      mypage.setInt(currentpos, getLastRecordPosition());
      setLastRecordPosition(currentpos);
      currentpos += INT_SIZE;
   }

   private int getLastRecordPosition() {
      return mypage.getInt(LAST_POS);
   }

   private void setLastRecordPosition(int pos) {
      mypage.setInt(LAST_POS, pos);
   }
}
```

Figure 13-6: The code for the SimpleDB class *LogMgr*

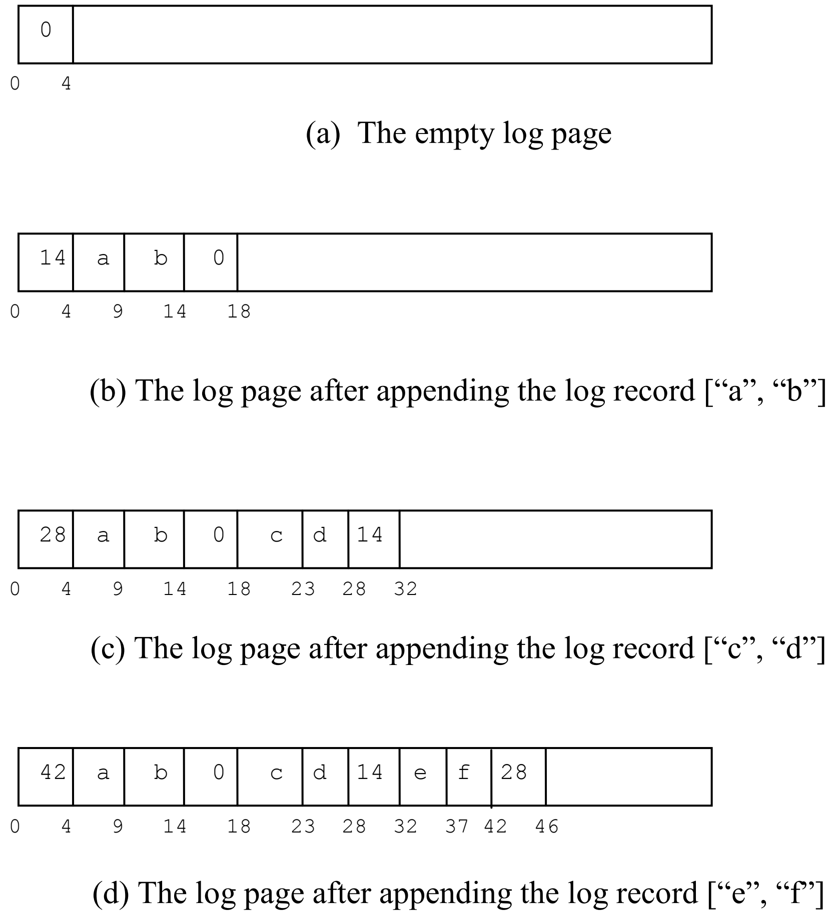
Recall that an LSN identifies a log record. For example, a typical LSN might consist of the block number of a log record and its starting location within that block. SimpleDB simplifies things and defines an LSN to be just the block number of the log record. That is, all log records in a block are assigned the same LSN. This simplification does not reduce correctness, but it does reduce efficiency a bit. See Exercise 13.11.

The method *flush* compares the current block number against the specified LSN. If the specified LSN is smaller, then the desired log record must have already been written to disk; otherwise, *mypage* is written to disk.

The *append* method calculates the size of the log record in order to determine if it will fit in the current page. If not, then it writes the current page to disk and calls *appendNewBlock* to clear the page and append the now-empty page to the log file. This strategy is slightly different from the algorithm of Figure 13-3; namely, the log manager extends the log file by appending an empty page to it, instead of extending the file by appending a full page. This strategy is simpler to implement, because it allows *flush* to assume that the block is already on disk.

Once the *append* method is sure that the log record will fit in the current page, it adds each value of the record to the page, and calls *finalizeRecord*. This method stores an integer after the new record that points to the location of the previous record; the result is that the records in the page will be chained backwards. In addition, the first integer in the page contains a pointer to the beginning of the chain – that is, the end of the last record.

For example of how this chain is created, consider Figure 13-7. This figure depicts the contents of the log page at four different times: when the page is first initialized, and after each of the three records of Figure 13-5 have been written. The page of Figure 13-7(a) contains only the value 0. This value denotes that the last record in the page ends at location 0 (i.e. it points to itself), which means that there are no records in the page.



(a) The empty log page

(b) The log page after appending the log record [“a”, “b”]

(c) The log page after appending the log record [“c”, “d”]

(d) The log page after appending the log record [“e”, “f”]

Figure 13-7: Appending three log records to an empty log page

Figure 13-7(b) depicts the page when it contains the log record [“a”, “b”]. Recall that each of these 1-character strings is 5 bytes long: 4 bytes for the length, and one byte for the character. The value 0 after the log record denotes (again) that the next record in the chain ends at location 0, which means that it is the last record in the chain. The value 14 at the beginning of the page denotes that the first record ends at location 14.

Figure 13-7(c) and 13-7(d) depict the page after the addition of the other two log records, and are similar to 13-7(b).

The class *LogIterator*

The method *iterator* in *LogMgr* flushes the log (in order to ensure that the entire log is on disk) and then returns a *LogIterator* object. The class *LogIterator* implements the iterator; its code appears in Figure 13-8. A *LogIterator* object allocates a single page to hold the contents of the log blocks as they are accessed. The constructor positions the iterator after the last record in the last block of the log. The method *next* uses the pointer chain to re-position the iterator to the previous record, moving to the next earlier log block if necessary, and returns an *BasicLogRecord* object corresponding to that position in the page.

```
class LogIterator implements Iterator<BasicLogRecord> {
   private Block blk;
   private Page pg = new Page();
   private int currentrec;

   LogIterator(Block blk) {
      this.blk = blk;
      pg.read(blk);
      currentrec = pg.getInt(LogMgr.LAST_POS);
   }

   public boolean hasNext() {
      return currentrec>0 || blk.number()>0;
   }

   public BasicLogRecord next() {
      if (currentrec == 0)
         moveToNextBlock();
      currentrec = pg.getInt(currentrec);
      return new BasicLogRecord(pg, currentrec+INT_SIZE);
   }

   public void remove() {
      throw new UnsupportedOperationException();
   }

   private void moveToNextBlock() {
      blk = new Block(blk.fileName(), blk.number()-1);
      pg.read(blk);
      currentrec = pg.getInt(LogMgr.LAST_POS);
   }
}
```

Figure 13-8: The code for the SimpleDB class *LogIterator*

The class *BasicLogRecord*

The code for the class *BasicLogRecord* appears in Figure 13-9. Its constructor receives the page containing the log record and the location of the first value in it. The methods *nextInt* and *nextString* read the value at that location in the page, and then increment the location by the size of the value.

```
public class BasicLogRecord {
   private Page pg;
   private int pos;

   public BasicLogRecord(Page pg, int pos) {
      this.pg = pg;
      this.pos = pos;
   }

   public int nextInt() {
      int result = pg.getInt(pos);
      pos += INT_SIZE;
      return result;
   }

   public String nextString() {
      String result = pg.getString(pos);
      pos += STR_SIZE(result.length());
      return result;
   }
}
```

Figure 13-9: The code for the SimpleDB class *BasicLogRecord*

13.4 Managing User Data

The log file is used in a limited, well-understood way. The log manager can therefore carefully fine-tune the way that it manages memory; in SimpleDB for example, the log manager needs only a single dedicated page to perform its job optimally.

JDBC applications, on the other hand, access their data completely unpredictably. There is no way to know which block an application will request next, and whether it will ever access a previous block again. And even after an application is completely finished with its blocks, we can't know whether another application will access any of those blocks in the near future. This section describes how the database system can efficiently manage memory in this situation.

13.4.1 The buffer manager

> *The buffer manager is the portion of the database system responsible for the pages that hold user data.*

The buffer manager allocates a fixed set of pages, called the *buffer pool*. As mentioned in the beginning of this chapter, the buffer pool should fit into the computer's physical memory, and these pages should come from the I/O buffers held by the operating system.

In order to access a block, a client interacts with the buffer manager according to the protocol given in Figure 13-10.

1. The client asks the buffer manager to *pin* a page from the buffer pool to that block.
2. The client accesses the contents of the page as much as it desires.
3. When the client is done with the page, it tells the buffer manager to *unpin* it.

Figure 13-10: The protocol for accessing a disk block

We say that a page is *pinned* if some client is currently pinning it; otherwise the page is *unpinned*. The buffer manager is obligated to keep a page available to its clients for as long as it is pinned. Conversely, once a page becomes unpinned, the buffer manager is allowed to assign it to another block.

A pinned buffer page is in use by its clients.
An unpinned buffer page is available for reuse.

When a client asks the buffer manager to pin a page to a block, the buffer manager will encounter one of these four possibilities:

- The contents of the block is in some page in the buffer, and:
    - the page is pinned, or
    - the page is unpinned.
- The contents of the block is not currently in any buffer, and:
    - there exists at least one unpinned page in the buffer pool, or
    - all pages in the buffer pool are pinned.

The first case occurs when one or more clients are currently accessing the contents of the block. Since a page can be pinned by multiple clients, the buffer manager simply adds another pin to the page, and returns the page to the client. Each client that is pinning the page is free to concurrently read and modify its values. The buffer manager is not concerned about potential conflicts that may occur; instead, that is the job of the concurrency manager (as we shall see in Chapter 14).

The second case occurs when the client(s) that were using the buffer have now finished with it, but the buffer has not yet been reused. Since the contents of the block are still in the buffer page, the buffer manager can reuse the page by simply pinning it and returning it to the client.

The third case requires the buffer manager to read the block from disk into a buffer page. Several steps are involved. The buffer manager must first select a page to reuse; this page must be unpinned (because the pinned pages are still being used by clients). Second, if the selected page has been modified, then the buffer manager must first write the page contents back to disk; this action is called *flushing* the page. Finally, the block can be read into the selected page, and the page can be pinned.

The fourth case occurs when the buffers are heavily used, such as in the query-processing algorithms of Chapter 23. In this case, the buffer manager cannot satisfy the client request. The best solution is for the buffer manager to place the client on a wait list. When a buffer becomes unpinned, the buffer manager can pin that page to the block, and the client can continue.

13.4.2 Buffers

Each page in the buffer pool has associated status information, such as whether it is pinned, and if so, what block it is pinned to. A *buffer* is the object that contains this information. Every page in the buffer pool has its own buffer.

Each buffer observes the changes to its page, and is responsible for writing its modified page to disk. Just as with the log, a buffer can reduce disk accesses if it can delay writing its page. For example if the page is modified several times, then it is more efficient to write the page once, after all modifications have been made. A reasonable strategy therefore is to have the buffer postpone writing its page to disk until the page is unpinned.

Actually, the buffer can wait even longer than that. Suppose a modified page becomes unpinned, but is not written to disk. If the page gets pinned again to the same block (as in the second case above), the client will see the modified contents, exactly as it had been left. This has the same effect as if the page had been written to disk and then read back, but without the disk accesses. In a sense, the buffer's page acts as the in-memory version of its disk block. Any client wishing to use the block will simply be directed to the buffer page, which the client can read or modify without any disk access occurring. There are only two reasons why a buffer will ever need to write a modified page to disk: either the page is being replaced because the buffer is getting pinned to a different block (as in the third case above); or the recovery manager needs to write its contents to disk in order to guard against a possible system crash.

13.4.3 Buffer Replacement Strategies

The pages in the buffer pool begin unallocated. As pin requests arrive, the buffer manager primes the buffer pool by assigning the requested blocks to unallocated pages. Once all pages have been allocated, the buffer manager will have to begin replacing pages. The buffer manager can choose to replace any page in the buffer pool, provided that it is unpinned.

If the buffer manager needs to replace a page and all buffers are pinned, then the requesting client must wait. Consequently, each client has the responsibility to "be a good citizen" and unpin a buffer as soon as it is no longer needed. A malicious client could bring the system to its knees by simply pinning all the buffers and leaving them pinned.[†]

If more than one buffer page is unpinned, then the buffer manager must decide which one to replace. This choice affects the number of disk accesses that will be performed. For example, the worst choice would be to replace the page that will be accessed next, because the buffer manager would then have to immediately replace another page. It turns out that the best choice is to always replace the page that will be unused for the longest amount of time.

Since the buffer manager cannot predict which pages will be accessed next, it is forced to guess. Here, the buffer manager is in almost exactly the same situation as the OS when it swaps pages in virtual memory. However there is one big difference: Unlike the OS, the buffer manager *knows* whether a page is currently being used or not, because the unused pages are exactly the unpinned pages. The burden of not being able to replace pinned pages turns out to be a blessing. Clients, by pinning pages responsibly, keep the buffer manager from making the really bad guesses. The buffer replacement strategy only has to choose from among the currently-unwanted pages, which is far less critical.

Given the set of unpinned pages, the buffer manager needs to decide which of those pages will not be needed for the longest amount of time. For example, a database usually has several pages (such as the catalog files of Chapter 16) that are used constantly throughout the lifetime of the database. The buffer manager ought to avoid replacing such pages, since they will almost certainly be re-pinned fairly soon.

As with OS virtual memory, there are various replacement strategies that try to approximate making the best guess. We shall briefly consider four general replacement strategies: *Naïve*, *FIFO*, *LRU*, and *Clock*.

Figure 13-11 introduces an example that will allow us to compare the behavior of these replacement algorithms. Part (a) gives a sequence of operations that pin and unpin five blocks of a file; and part (b) depicts the resulting state of the buffer pool, assuming that it contains four buffers. The only page replacement occurred when the fifth block (i.e. block 50) was pinned. However, since only one buffer was unpinned at that time, the buffer manager had no choice. In other words, the buffer pool would look like Figure13-11(b), regardless of the page replacement strategy.

Each buffer in Figure 13-11(b) holds three pieces of information: its block number, the time when it was read into the buffer, and the time when it became unpinned. The times in the figure correspond to the position of the operation in Figure 13-11(a).

[†] Although a JDBC client does not have direct access to buffers, it sometimes can still be malicious. For example, it can create (and not close) many result sets, each of which is accessing a different block; this forces the buffer manager to use a separate buffer for each block.

```
pin(10); pin(20); pin(30); pin(40); unpin(20);
pin(50); unpin(40); unpin(10); unpin(30); unpin(50);
```

(a) A sequence of ten *pin*/*unpin* operations

Buffer:	0	1	2	3
block#	10	50	30	40
time read in	1	6	3	4
time unpinned	8	10	9	7

(b) The resulting state of the buffer pool

Figure 13-11: The effect of some *pin*/*unpin* operations on a pool of 4 buffers

The buffers of Figure 13-11(b) are all unpinned. Suppose now that the buffer manager receives two more pin requests:

```
pin(60); pin(70);
```

The buffer manager will need to replace two buffers. All of the buffers are available; which ones should it choose? Each of the following replacement algorithms will give a different answer.

The naïve strategy

The simplest replacement strategy is to traverse the buffer pool sequentially, replacing the first unpinned buffer found. Using the example of Figure 13-11, block 60 will be assigned to buffer 0, and block 70 will be assigned to buffer 1.

This strategy is easy to implement, but has little else to recommend it. For example, consider again the buffers of Figure 13-11, and suppose a client repeatedly pins and unpins blocks 60 and 70, like this:

```
pin(60); unpin(60); pin(70); unpin(70); pin(60); unpin(60); pin(70);...
```

The naïve replacement strategy will use buffer 0 for both blocks, which means that the blocks will need to be read in from disk each time they are pinned. The problem is that the buffer pool is not evenly utilized. Had the replacement strategy chosen two different buffers for blocks 60 and 70, then the blocks would have been read from disk only once each – which is a tremendous improvement in efficiency.

The FIFO strategy

The naïve strategy chooses a buffer based only on convenience. The FIFO strategy tries to be more intelligent, by choosing the buffer that was least recently replaced – that is, the page that has been sitting in the buffer pool the longest. This strategy usually works better than the naïve strategy, because older pages are less likely to be needed than more recently-fetched pages. In Figure 13-11, the oldest pages are the ones with the smallest values for "time read in". Thus block 60 would get assigned to buffer 0, and block 70 would get assigned to buffer 2.

FIFO is a reasonable strategy, but it does not always make the right choice. For example, a database often has frequently-used pages, such as catalog pages. Since these pages are used by nearly every client, it makes sense to not replace them if at all possible. However, these pages will eventually become the oldest pages in the pool, and the FIFO strategy will choose them for replacement.

The FIFO replacement strategy can be implemented in two ways. One way is to have each buffer hold the time when its page was last replaced, as in Figure 13-11(b). The replacement algorithm would then scan the buffer pool, choosing the unpinned page having the earliest replacement time. A second, more efficient way would be for the buffer manager to keep a list of pointers to its buffers, ordered by replacement time. The replacement algorithm searches the list; the first unpinned page found is replaced, and the pointer to it is moved to the end of the list.

The LRU strategy

The FIFO strategy bases its replacement decision on when a page was *added* to the buffer pool. A similar strategy would be to make the decision based on when a page was last *accessed*, the rationale being that a page that has not been used in the near past will also not be used in the near future. This strategy is called LRU, which stands for *least recently used*. In the example of Figure 13-11, the "time unpinned" value corresponds to when the buffer was last used. Thus block 60 would be assigned to buffer 3, and block 70 would be assigned to buffer 0.

The LRU strategy tends to be an effective general-purpose strategy, and avoids replacing commonly-used pages. Both of the implementation options for FIFO can be adapted to LRU. The only change that must be made is that the buffer manager must update the timestamp (for the first option) or update the list (for the second option) each time a page becomes *unpinned*, instead of when it gets replaced.

The clock strategy

This strategy is an interesting combination of the above strategies that has an easy and straightforward implementation. As in the naïve strategy, the replacement algorithm scans through the buffer pool, choosing the first unpinned page it finds. The difference is that the algorithm always starts its scan at the page after the previous replacement. If you visualize the buffer pool as forming a circle, then the replacement algorithm scans the pool like the hand of an analog clock, stopping when a page is replaced and starting when another replacement is required.

The example of Figure 13-11(b) does not indicate the clock position. But the last replacement it made was buffer 1, which means that the clock is positioned immediately after that. Thus block 60 will be assigned to buffer 2, and block 70 will be assigned to buffer 3.

The clock strategy attempts to use the buffers as evenly as possible. If a page is pinned, the clock strategy will skip past it and not consider it again until it has examined all other buffers in the pool. This feature gives the strategy an LRU flavor. The idea is that if a page is frequently used, there is a high probability that it will be pinned when its turn for replacement arrives. If so, then it is skipped over and given "another chance".

13.5 The SimpleDB Buffer Manager

This section examines the buffer manager of the SimpleDB database system. Section 13.5.1 covers the buffer manager's API, and gives examples of its use. Section 13.5.2 then shows how these classes can be implemented in Java.

13.5.1 An API for the buffer manager

The SimpleDB buffer manager is implemented by the package *simpledb.buffer*. This package exposes the two classes *BufferMgr* and *Buffer*, as well as the interface *PageFormatter*; their API appears in Figure 13-12.

BufferMgr

```
    public BufferMgr(int numbuffs);
    public Buffer pin(Block blk);
    public Buffer pinNew(String filename, PageFormatter fmtr);
    public void   unpin(Buffer buff);
    public void   flushAll(int txnum);
    public int    available();
```

Buffer

```
    public int    getInt(int offset);
    public String getString(int offset);
    public void   setInt(int offset, int val, int txnum, int lsn);
    public void setString(int offset, String val, int txnum, int lsn);
    public Block  block();
```

PageFormatter

```
    public void format(Page p);
```

Figure 13-12: The API for the SimpleDB buffer manager

Each database system has one *BufferMgr* object, which is created during system startup. The class *server.SimpleDB* creates the object by passing the size of the buffer pool into the constructor; its static method *buffMgr* returns the created object.

A *BufferMgr* object has methods to pin and unpin buffers. The method *pin* returns a buffer pinned to the specified block, and the *unpin* method unpins the buffer. A client can read the contents of a pinned buffer by calling its *getInt* and *getString* methods. The code fragment of Figure 13-13 provides a simple example of buffer usage. The code pins a buffer to block 3 of file *junk*, gets values from offsets 392 and 20, and then unpins the buffer.

```
SimpleDB.init("studentdb");
BufferMgr bm = SimpleDB.bufferMgr();
Block  blk   = new Block("junk", 3);
Buffer buff  = bm.pin(blk);
int    n = buff.getInt(392);
String s = buff.getString(20);
bm.unpin(buff);
System.out.println("The values are " + n + " and " + s);
```

Figure 13-13: Reading values from a buffer

A client modifies the values in a block by calling the buffer's *setInt* and *setString* methods. Before doing so, however, the client must first append an appropriate record to the log and obtain its LSN. The client passes this LSN (and a transaction identifier) into the *setInt* and *SetString* methods. The buffer saves these values, and will use them when it needs to force the log record to disk.

The code fragment of Figure 13-14 illustrates buffer modification, by incrementing the integer at offset 392 of the buffer. Before it calls *setInt*, the code creates a log record and appends it to the log; the LSN of that log record is passed into the *setInt* method.

```
SimpleDB.init("studentdb");
BufferMgr bm = SimpleDB.bufferMgr();
Block  blk  = new Block("junk", 3);
Buffer buff = bm.pin(blk);
int n = buff.getInt(392);

LogMgr lm = SimpleDB.logMgr();
int mytxnum = 1;  // assume we are transaction #1
Object[] logrec = new Object[] {"junk", 3, 392, n};
int lsn = lm.append(logrec);

buff.setInt(392, n+1, mytxnum, lsn);
bm.unpin(buff);
```

Figure 13-14: Writing a value to a buffer

The buffer manager doesn't care what log record gets created, and so the log record in Figure 13-14 could have been anything. But just for fun, the code creates a log record similar to what SimpleDB would create. The log record contains four values: the filename, block number, and offset of the modified value, plus the previous value. This information is sufficient to undo the modification if desired.

The *setInt* and *setString* methods modify an existing value in the block. So how do the initial values get created? SimpleDB assumes that each new block is *formatted* when it is appended to the file. The formatting is performed by the method *format* of a *PageFormatter* object. SimpleDB contains several kinds of block (such as index blocks and data blocks), and each kind of block has its own *PageFormatter* class. Figure 13-15 gives the code for an example formatter class that fills the block with multiple copies of the string "abc". This formatter code is not especially interesting. A more useful implementation of *PageFormatter* will be considered later, in Figure 15-12.

```
class ABCStringFormatter implements PageFormatter {
      public void format(Page p) {
            int recsize = STR_SIZE("abc");
            for (int i=0; i+recsize <= BLOCK_SIZE; i+=recsize)
                  p.setString(i, "abc");
      }
}
```

Figure 13-15: A simple (but utterly useless) page formatter

The *BufferMgr* method *pinNew* is responsible for creating new file blocks. The method takes two arguments: the name of the file, and a page formatter. The method allocates a buffer page for the new block, pins and formats the page, appends the formatted page to the file, and returns the buffer to the client. The code of Figure 13-16 illustrates the use of *pinNew*. The code adds a new block (filled with "abc"s) to the file *junk*; it then reads the first value from it, and obtains the block number from the buffer.

```
SimpleDB.init("studentdb");
BufferMgr bm = SimpleDB.bufferMgr();
PageFormatter pf = new ABCStringFormatter();
Buffer buff = bm.pinNew("junk", pf);
String s    = buff.getString(0);  //will be "abc"
int blknum  = buff.block().number();
bm.unpin(buff);
System.out.println("The first value in block " + blknum +
                   " is " + s);
```

Figure 13-16: Adding a new block to the file *junk*

The class *BufferMgr* also contains two other methods. The method *flushAll* is used to clean up a specified transaction. It flushes each buffer modified by that transaction, thereby making its changes permanent. The method *available* returns the number of currently-unpinned buffers. This value is used by the query evaluation algorithms of Chapter 23.

Note how the buffer manager hides the actual disk accesses from its clients. A client has no idea exactly how many disk accesses occur on its behalf, and when they occur. Disk reads can occur only during calls to *pin*; in particular, a disk read will occur only when the specified block is not currently in a buffer. Disk writes will occur during calls to *pin* and *pinNew*, provided that the replaced buffer had been modified, and to *flushAll*, if the transaction has any modified buffers.

Suppose that the database system has a lot of clients, all of whom are using a lot of buffers. It is possible for all of the buffers in the buffer pool to be pinned. In this case, the buffer manager cannot immediately satisfy a *pin* or *pinNew* request. Instead, it places the calling thread on a wait list until a buffer becomes available, and then returns the buffer. In other words, the client will not be aware of the buffer contention; the client will only notice that the program has slowed down.

There is one situation where buffer contention can cause a serious problem. Consider a scenario where clients A and B each need two buffers, but only two buffers are available. Suppose client A pins the first buffer. There is now a race for the second buffer. If client A gets it before client B, then B will be added to the wait list. Client A will eventually finish and unpin the buffers, at which time client B can pin them. This is a good scenario. Now suppose that client B gets the second buffer before client A. Then both A and B will be on the wait list. If these are the only two clients in the system, then no buffers will ever get unpinned and both A and B will be on the wait list forever. This is a bad scenario. Clients A and B are said to be *deadlocked*.

In a real database system with thousands of buffers and hundreds of clients, this kind of deadlock is highly unlikely. Nevertheless, the buffer manager must be prepared to deal with the possibility. The solution taken by SimpleDB is to keep track of how long a client has been waiting for a buffer. If it has waited too long (say, 10 seconds), then the buffer manager assumes that the client is in deadlock and throws an exception of type *BufferAbortException*. The client is responsible for handling the exception, typically by rolling back the transaction and possibly restarting it.

13.5.2 Implementing the buffer manager

The class *Buffer*

Figure 13-17 contains the code for class *Buffer*.

```
public class Buffer {
   private Page contents = new Page();
   private Block blk = null;
   private int pins = 0;
   private int modifiedBy = -1;
   private int logSequenceNumber = -1;

   public int getInt(int offset) {
      return contents.getInt(offset);
   }

   public String getString(int offset) {
      return contents.getString(offset);
   }

   public void setInt(int offset, int val, int txnum, int lsn) {
      modifiedBy = txnum;
      if (lsn >= 0)
         logSequenceNumber = lsn;
      contents.setInt(offset, val);
   }

   public void setString(int offset, String val, int txnum, int lsn) {
      modifiedBy = txnum;
      if (lsn >= 0)
         logSequenceNumber = lsn;
      contents.setString(offset, val);
   }

   public Block block() {
      return blk;
   }

   void flush() {
      if (modifiedBy >= 0) {
         SimpleDB.logMgr().flush(logSequenceNumber);
         contents.write(blk);
      }
      modifiedBy = -1;
   }

   void pin() {
      pins++;
   }

   void unpin() {
      pins--;
   }

   boolean isPinned() {
      return pins > 0;
   }
   boolean isModifiedBy(int txnum) {
      return txnum == modifiedBy;
   }

   void assignToBlock(Block b) {
      flush();
      blk = b;
      contents.read(blk);
      pins = 0;
   }

   void assignToNew(String filename, PageFormatter fmtr) {
      flush();
      fmtr.format(contents);
      blk = contents.append(filename);
      pins = 0;
   }
}
```

Figure 13-17: The code for the SimpleDB class *Buffer*

A *Buffer* object keeps track of four kinds of information about its page:

- *A logical reference to the block assigned to its page*. If no block is assigned, then the value is null.
- *The number of times the page is pinned*. The pin count is incremented on each pin, and decremented on each unpin.
- *An integer indicating if the page has been modified*. A value of -1 indicates that the page has not been changed; otherwise the integer identifies the transaction that made the change.
- *Log information*. If the page has been modified, then the buffer holds the LSN of the most recent log record. LSN values are never negative. If a client calls *setInt* or *setString* with a negative LSN, it indicates that a log record was not generated for that update.

The method *flush* ensures that the buffer's page has the same values as its assigned disk block. If the buffer has not been modified, then the method need not do anything. If it has been modified, then the method first calls *LogMgr.flush* to ensure that the corresponding log record is on disk; then it writes the page to disk.

The methods *assignToBlock* and *assignToNew* associate the buffer with a disk block. In both methods, the buffer is first flushed, so that any modifications to the previous block are preserved. Method *assignToBlock* associates the buffer with the specified block, reading its contents from disk. Method *assignToNew* performs the formatting, appends the page to the file, and then associates the buffer with that new block.

The class *BasicBufferMgr*

In SimpleDB, the code for the buffer manager is divided between two classes: *BufferMgr* and *BasicBufferMgr*. The class *BasicBufferMgr* manages the buffer pool; the class *BufferMgr* manages the wait list. The code for *BasicBufferMgr* appears in Figure 13-18.

```
class BasicBufferMgr {
   private Buffer[] bufferpool;
   private int numAvailable;

   BasicBufferMgr(int numbuffs) {
      bufferpool = new Buffer[numbuffs];
      numAvailable = numbuffs;
      for (int i=0; i<numbuffs; i++)
         bufferpool[i] = new Buffer();
   }

   synchronized void flushAll(int txnum) {
      for (Buffer buff : bufferpool)
         if (buff.isModifiedBy(txnum))
            buff.flush();
   }

   synchronized Buffer pin(Block blk) {
      Buffer buff = findExistingBuffer(blk);
      if (buff == null) {
         buff = chooseUnpinnedBuffer();
         if (buff == null)
            return null;
         buff.assignToBlock(blk);
      }
      if (!buff.isPinned())
         numAvailable--;
      buff.pin();
      return buff;
   }

   synchronized Buffer pinNew(String filename, PageFormatter fmtr) {
      Buffer buff = chooseUnpinnedBuffer();
      if (buff == null)
         return null;
      buff.assignToNew(filename, fmtr);
      numAvailable--;
      buff.pin();
      return buff;
   }

   synchronized void unpin(Buffer buff) {
      buff.unpin();
      if (!buff.isPinned())
         numAvailable++;
   }

   int available() {
      return numAvailable;
   }

   private Buffer findExistingBuffer(Block blk) {
      for (Buffer buff : bufferpool) {
         Block b = buff.block();
         if (b != null && b.equals(blk))
            return buff;
      }
      return null;
   }

   private Buffer chooseUnpinnedBuffer() {
      for (Buffer buff : bufferpool)
         if (!buff.isPinned())
            return buff;
      return null;
   }
}
```

Figure 13-18: The code for the SimpleDB class *BasicBufferMgr*

The method *pin* assigns a buffer to the specified block. Its algorithm has two parts. The first part, *findExistingBuffer*, tries to find a buffer that is already assigned to the specified block. The buffer is returned if found. Otherwise the second part of the algorithm, *chooseUnpinnedBuffer*, uses naïve replacement to choose an unpinned buffer. That buffer's *assignToBlock* method is called, which handles the writing of the existing page to disk (if necessary) and the reading of the new page from disk. The method returns *null* if it cannot find an unpinned buffer.

The method *pinNew* assigns a buffer to a new block in the specified file. It is similar to *pin*, except that it immediately calls *chooseUnpinnedBuffer*. (There is no point in calling *findExistingBuffer*, because the block doesn't exist yet.) It calls *assignToNew* on the buffer it finds, which causes a new block to be created. The method also returns *null* if it cannot find an unpinned buffer.

The class *BufferMgr*

The class *BasicBufferMgr* performs all of the buffer manager functions, except one: it returns null when it cannot find an unpinned buffer. The class *BufferMgr* wraps *BasicBufferMgr*, redefining *pin* and *pinNew*; its code appears in Figure 13-19. Instead of returning *null* when a buffer is not available, these methods now call the Java method *wait*. In Java, every object has a wait list. The object's *wait* method interrupts the execution of the calling thread and places it on that list. In Figure 13-19, the thread will be removed from the list and readied for execution when one of two conditions occurs:

- Another thread calls *notifyAll* (which occurs from a call to *unpin*); or
- MAX_TIME milliseconds have elapsed, which means that it has been waiting too long.

When a waiting thread resumes, it continues in its loop, trying to obtain a buffer. The thread will keep getting placed back on the wait list until either it gets the buffer or it has exceeded its time limit.

```
public class BufferMgr {
   private static final long MAX_TIME = 10000; // 10 seconds
   private BasicBufferMgr bufferMgr;

   public BufferMgr(int numbuffers) {
      bufferMgr = new BasicBufferMgr(numbuffers);
   }

   public synchronized Buffer pin(Block blk) {
      try {
         long timestamp = System.currentTimeMillis();
         Buffer buff = bufferMgr.pin(blk);
         while (buff == null && !waitingTooLong(timestamp)) {
            wait(MAX_TIME);
            buff = bufferMgr.pin(blk);
         }
         if (buff == null)
            throw new BufferAbortException();
         return buff;
      }
      catch(InterruptedException e) {
         throw new BufferAbortException();
      }
   }

   public synchronized Buffer pinNew(String filename,
                                     PageFormatter fmtr) {
      try {
         long timestamp = System.currentTimeMillis();
         Buffer buff = bufferMgr.pinNew(filename, fmtr);
         while (buff == null && !waitingTooLong(timestamp)) {
            wait(MAX_TIME);
            buff = bufferMgr.pinNew(filename, fmtr);
         }
         if (buff == null)
            throw new BufferAbortException();
         return buff;
      }
      catch(InterruptedException e) {
         throw new BufferAbortException();
      }
   }

   public synchronized void unpin(Buffer buff) {
      bufferMgr.unpin(buff);
      if (!buff.isPinned())
         notifyAll();
   }
   public void flushAll(int txnum) {
      bufferMgr.flushAll(txnum);
   }

   public int available() {
      return bufferMgr.available();
   }

   private boolean waitingTooLong(long starttime) {
      return System.currentTimeMillis() - starttime > MAX_TIME;
   }
}
```

Figure 13-19: The code for the SimpleDB class *BufferMgr*

The *unpin* method calls the basic buffer manager to unpin the specified buffer; it then checks to see if that buffer is still pinned. If not, then *notifyAll* is called to remove all client threads from the wait list. Those threads will fight for the buffer; whichever is scheduled first will win. When one of the other threads is scheduled, it may find that all buffers are still allocated; if so, it will be placed back on the wait list.

13.6 Chapter Summary

- A database system must strive to minimize disk accesses. It therefore carefully manages the in-memory pages that it uses to hold disk blocks. The database components that manage these pages are the *log manager* and the *buffer manager*.

- The log manager is responsible for saving log records in the log file. Because log records are always appended to the log file and are never modified, the log manager can be very efficient. It only needs to allocate a single page, and has a simple algorithm for writing that page to disk as few times as possible.

- The buffer manager allocates several pages, called the *buffer pool*, to handle user data. The buffer manager *pins* and *unpins* buffers to disk blocks, at the request of clients. A client accesses a buffer's page after it is pinned, and unpins the buffer when finished.

- A modified buffer will get written to disk in two circumstances: when the page is being replaced, and when the recovery manager needs it to be on disk.

- When a client asks to pin a buffer to a block, the buffer manager chooses the appropriate buffer. If that block is already in a buffer, then that buffer is used; otherwise, the buffer manager replaces the contents of an existing buffer.

- The algorithm that determines which buffer to replace is called the *buffer replacement strategy*. Four interesting replacement strategies are:
    - *Naïve*: Choose the first unpinned buffer it finds.

- *FIFO*: Choose the unpinned buffer whose contents were replaced least recently.
- *LRU*: Choose the unpinned buffer whose contents were unpinned least recently.
- *Clock*: Scan the buffers sequentially from the last replaced buffer, and choose the first unpinned buffer found.

13.7 Suggested Reading

The article [Effelsberg et al. 1984] contains a well-written and comprehensive treatment of buffer management that extends many of the ideas in this chapter. Chapter 13 of [Gray and Reuter 1993] contains an in-depth discussion of buffer management, illustrating their discussion with a C-based implementation of a typical buffer manager.

Several researchers have investigated how to make the buffer manager itself more intelligent. The basic idea is that a buffer manager can keep track of the pin requests of each transaction. If it detects a pattern (say, the transaction repeatedly reads the same N blocks of a file), it will try to avoid replacing those pages, even if they are not pinned. The article [Ng et al 1991] describes the idea in more detail, and provides some simulation results.

13.8 Exercises

CONCEPTUAL EXERCISES

13.1 The code for *LogMgr.iterator* calls *flush*. Is this call necessary? Explain.

13.2 Explain why the method *BasicBufferMgr.pin* is synchronized. What problem could occur if it wasn't?

13.3 Can more than one buffer ever be assigned to the same block? Explain.

13.4 The buffer replacement strategies we have considered do not distinguish between modified and unmodified pages when looking for an available buffer. A possible improvement is for the buffer manager to always replace an unmodified page whenever possible.
a) Give one reason why this suggestion could reduce the number of disk accesses made by the buffer manager.
b) Give one reason why this suggestion could increase the number of disk accesses made by the buffer manager.
c) Do you think this is a good strategy? Explain.

13.5 Another possible buffer replacement strategy is *least recently modified*: the buffer manager chooses the modified buffer having the lowest LSN. Explain why such a strategy might be worthwhile.

13.6 Suppose that a buffer has been modified several times without being written to disk. The buffer saves only the LSN of the most recent change, and sends only this LSN to the log manager when the page is finally flushed. Explain why the buffer doesn't need to send the other LSNs to the log manager.

13.7 Consider the example pin/unpin scenario of Figure 13-11(a), together with the additional operations *pin(60); pin(70)*. For each of the four replacement strategies given in the text, draw the state of the buffers, assuming that the buffer pool contains 5 buffers.

13.8 Starting from the buffer state of Figure 13-11(b), give a scenario in which:
a) the FIFO strategy requires the fewest disk accesses.
b) the LRU strategy requires the fewest disk accesses.
c) the clock strategy requires the fewest disk accesses.

13.9 Suppose that two different clients each want to pin the same block, but are placed on the wait list because no buffers are available. Consider the implementation of the SimpleDB class *BufferMgr*. Show that when a single buffer becomes available, both clients will be able to use the same buffer.

13.10 Consider the adage "Virtual is its own reward." Comment on the cleverness of the pun, and discuss its applicability to the buffer manager.

PROGRAMMING EXERCISES

13.11 The SimpleDB log manager implements the LSN of a log record as its block number in the log file.
a) Explain why this implementation of an LSN does not uniquely identify the log records.
b) Explain why this implementation may cause the method *LogMgr.flush* to write the log manger's page to disk unnecessarily.
c) Explain a better way to implement an LSN.
d) Modify the code (and the implementation of an LSN) so that the page is written to disk only if absolutely necessary.

13.12 The SimpleDB log manager allocates its own page and writes it explicitly to disk. Another design option is for it to pin a buffer to the last log block, and let the buffer manager handle the disk accesses.
a) Work out a design for this option. What are the issues that need to be addressed? Is it a good idea?
b) Modify SimpleDB to implement your design.

13.13 Each *LogIterator* object allocates a page to hold the log blocks it accesses.
a) Explain why using a buffer instead of a page would be much more efficient.
b) Modify the code to use a buffer instead of a page. How should the buffer get unpinned?

13.14 In Section 3.4.3 we mentioned how a JDBC program could maliciously pin all of the buffers in the buffer pool.
a) Write a JDBC program to pin all of the buffers of the SimpleDB buffer pool. What happens when all of the buffers are pinned?
b) The Derby database system does buffer management differently than SimpleDB. When a JDBC client requests a buffer, Derby pins the buffer, sends a copy of the buffer to the client, and unpins the buffer. Explain why your code will not be malicious to other Derby clients.
c) Derby avoids SimpleDB's problem by always copying pages from server to client. Explain the consequences of this approach. Do you prefer it to the SimpleDB approach?

13.15 Modify class *BasicBufferMgr* to implement each of the other replacement strategies described in this chapter.

13.16 Exercise 13.4 suggests a page replacement strategy that chooses unmodified pages over modified ones. Implement this replacement strategy.

13.17 Exercise 13.5 suggests a page replacement strategy that chooses the modified page having the lowest LSN. Implement this strategy.

13.18 One way to keep a rogue client from monopolizing all of the buffers is to allow each transaction to pin no more than a certain percentage (say, 10%) of the buffer pool. Implement and test this modification to the buffer manager.

13.19 The SimpleDB buffer manager traverses the buffer pool sequentially when searching for buffers. This search will be time-consuming when there are thousands of buffers in the pool. Modify the code, adding data structures (such as special-purpose lists and hash tables) that will improve the search times.

13.20 Write programs to unit test the log and buffer managers.

13.21 In Exercise 12.15 you were asked to write code that maintained statistics about disk usage. Extend this code to also give information about buffer usage.

14

TRANSACTION MANAGEMENT

and the packages in simpledb.tx

The buffer manager allows multiple clients to access the same buffer concurrently, arbitrarily reading and writing values from it. The result can be chaos: A page might have different (and even inconsistent) values each time a client looks at it, making it impossible for the client to get an accurate picture of the database. Or two clients can unwittingly overwrite the values of each other, thereby corrupting the database.

In this chapter we study the *recovery* and *concurrency managers*, whose jobs are to maintain order and ensure database integrity. Each client program gets structured as a sequence of transactions. The concurrency manager regulates the execution of these transactions so that they behave consistently. The recovery manager reads and writes records from the log, so that changes made by uncommitted transactions can be undone if necessary. This chapter covers the functionality of these managers, as well as the techniques used to implement them.

14.1 Transactions

14.1.1 Correct code that behaves incorrectly

Consider an airline reservation database, having two tables with the following schemas:

```
SEATS(FlightId, NumAvailable, Price)
CUST(CustId, BalanceDue)
```

Figure 14-1 contains JDBC code to purchase a ticket for a specified customer on a specified flight. Although this code has no bugs, various problems can occur when it is being used concurrently by multiple clients or if the server crashes. The following three scenarios illustrate these problems.

For the first scenario, suppose that both clients A and B run the JDBC code concurrently, with the following sequence of actions:

- Client A executes all of step 1, and is then interrupted.
- Client B executes to completion.
- Client A completes its execution.

In this case, both threads will use the same value for *numAvailable*. The result is that two seats will be sold, but the number of available seats will be decremented only once.

```
public void reserveSeat(Connection conn, int custId, int flightId)
                               throws SQLException {

   Statement stmt = conn.createStatement();
   String s;

// step 1: Get availability and price
   s  = "select NumAvailable, Price from SEATS " +
        "where FlightId = " + flightId;
   ResultSet rs = stmt.executeQuery(s);
   if (!rs.next()) {
      System.out.println("Flight doesn't exist");
      return;
   }
   int numAvailable = rs.getInt("NumAvailable");
   int price = rs.getInt("Price");
   rs.close();

   if (numAvailable == 0) {
      System.out.println("Flight is full");
      return;
   }

// step 2: Update availability
   int newNumAvailable = numAvailable - 1;
   s  = "update SEATS set NumAvailable = " + newNumAvailable +
        " where FlightId = " + flightId;
   stmt.executeUpdate(s);

// step 3: Get and update customer balance
   s = "select BalanceDue from CUST where CustID = " + custId;
   rs = stmt.executeQuery(s);
   int newBalance = rs.getInt("BalanceDue") + price;
   rs.close();

   s  =  "update CUST set BalanceDue = " + newBalance +
         " where CustId = " + custId;
   stmt.executeUpdate(s);
}
```

Figure 14-1: JDBC code to reserve a seat on a flight

For the second scenario, suppose that thread C is running the code, and the server crashes just after step two executes. In this case, the seat will be reserved, but the customer will not be charged for it.

For the third scenario, suppose that a client runs the code to completion, but the modified pages are not immediately written to disk due to buffering. If the server crashes (possibly several days later), then there is no way to know which of the pages (if any) were eventually written to disk. If the first update was written but not the second, then the customer receives a free ticket; if the second update was written but not the first, then the customer is charged for a non-existent ticket. And if neither page was written, then the entire interaction will be lost.

14.1.2 Properties of transactions

The above scenarios show how data can get lost or corrupted when client programs are able to run indiscriminately. Database systems solve this problem by forcing client programs to consist of *transactions*.

> *A transaction is a group of operations that behaves as a single operation. A transaction should satisfy the four ACID properties.*

The meaning of "as a single operation" can be characterized by the following so-called *ACID* properties: *Atomicity*, *Consistency*, *Isolation*, and *Durability*.

- Atomicity means that a transaction is "all or nothing". That is, either all of its operations succeed (the transaction *commits*), or they all fail (the transaction does a *rollback*).
- Consistency means that every transaction leaves the database in a consistent state. This implies that each transaction is a complete work unit that can be executed independently of other transactions.
- Isolation means that a transaction behaves as if it is the only thing running on the system. That is, if multiple transactions are running concurrently, then their result should be the same as if they were all executed serially in some order.
- Durability means that changes made by a committed transaction are guaranteed to be permanent.

Each of the problems illustrated in Section 14.1.1 results from some violation of the ACID properties. The first scenario violated the isolation property, because both clients read the same value for *numAvailable*, whereas in any serial execution the second client would have read the value written by the first. The second scenario violated atomicity, and the third scenario violated durability.

> *The atomicity and durability properties describe the proper behavior of the commit and rollback operations.*

In particular, a committed transaction must be durable, and an uncommitted transaction (either due to an explicit rollback or to a system crash) must have its changes completely undone. These features are the responsibility of the *recovery manager*, and are the topic of Section 14.3.

> *The consistency and isolation properties describe the proper behavior of concurrent clients.*

In particular, the database server must keep clients from conflicting with each other. A typical strategy is to detect when a conflict is about to occur, and to make one of the clients wait until that conflict is no longer possible. These features are the responsibility of the *concurrency manager*, and are the topic of Section 14.4.

14.2 Using Transactions in SimpleDB

Before we get into details about how the recovery and concurrency managers do their job, it will help to get a feel for how clients use transactions. In SimpleDB, every JDBC transaction has its own *Transaction* object; its API appears in Figure 14-2.

Transaction

```
public Transaction();
public void    commit();
public void    rollback();
public void    recover();

public void    pin(Block blk);
public void    unpin(Block blk);
public int     getInt(Block blk, int offset);
public String  getString(Block blk, int offset);
public void    setInt(Block blk, int offset, int val);
public void    setString(Block blk, int offset, String val);

public int     size(String filename);
public Block   append(String filename, PageFormatter fmtr);
```

Figure 14-2: The API for the SimpleDB transactions

The methods of *Transaction* fall into three categories. The first category consists of methods related to the transaction's lifespan. The constructor begins a new transaction, and the *commit* and *rollback* methods terminate it. The method *recover* performs a rollback on all uncommitted transactions.

The second category consists of methods to access a disk block. These methods are similar to those of the buffer manager. One important difference is that a transaction hides the existence of buffers from its client. When a client calls *pin* on a block, the transaction saves the buffer internally and does not return it to the client. When the client calls a method such as *getInt*, it passes in a *Block* reference. The transaction then finds the corresponding buffer, calls the buffer's *getInt* method, and passes the result back to the client. The transaction hides the buffer from the client so it can make the necessary calls to the concurrency and recovery managers. For example, the code for *setInt* will acquire the appropriate locks (for concurrency control) and write the value that is currently in the buffer to the log (for recovery) before modifying the buffer.

The third category consists of two methods related to the file manager. The method *size* reads the end of the file marker, and *append* modifies it; therefore these methods must call the concurrency manager to avoid potential conflicts.

The code fragment in Figure 14-3 illustrates a simple use of the *Transaction* methods.

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();
Block  blk  = new Block("junk", 3);
tx.pin(blk);
int n = tx.getInt(blk, 392);
String s = tx.getString(blk, 20);
tx.unpin(blk);
System.out.println("The values are " + n + " and " + s);

tx.pin(blk);
n = tx.getInt(blk, 392);   //Is this statement necessary?
tx.setInt(blk, 392, n+1);
tx.unpin(blk);

PageFormatter pf = new ABCStringFormatter();
Block newblk = tx.append("junk", pf);
tx.pin(newblk);
String s = tx.getString(newblk, 0);  //will be "abc"
tx.unpin(newblk);
int blknum  = newblk.number();
System.out.println("The first value in block " + blknum +
                   " is " + s);
tx.commit();
```

Figure 14-3: Using a SimpleDB transaction

This code consists of three sections, which perform the same tasks as the code of Figures 13-13 to 13-16. The first section pins block 3 of file *junk* and reads two values from it: an integer at offset 392 and a string at offset 20. The second section increments the integer at offset 392 of that block. The third section formats a new page, appends it to the file *junk*, and reads the value stored in offset 0. Finally, the transaction commits.

This code is considerably simpler than the corresponding code in Chapter 13, primarily because the *Transaction* object takes care of the buffering and the logging. But equally important is that the transaction also takes care of issues due to concurrent clients.

For example, observe that both the first and second sections of the code have the same call to *getInt*. Do we need that second call? Won't it always return exactly the same value as the first call? In the absence of concurrency control, the answer is "no". Suppose that this code gets interrupted after executing the first call but before executing the second one, and that another client executes during this interruption and modifies the value at offset 392. Then the second call will in fact return something different from the first call. (This is the "non-repeatable read" scenario of Section 8.2.3.) This scenario violates the isolation property of transactions, and so the *Transaction* object is responsible for making sure it will not occur. In other words, because of concurrency control, the second call to *getInt* really is unnecessary, as it should be.

14.3 Recovery Management

The *recovery manager* is the portion of the server that reads and processes the log. It has three functions: to write log records, to roll back a transaction, and to recover the database after a system crash. This section investigates these functions in detail.

14.3.1 Log records

In order to be able to roll back a transaction, the recovery manager saves information in the log about the transaction's activities. In particular, it writes a *log record* to the log each time a loggable activity occurs. There are four basic kinds of log record: *start records*, *commit records*, *rollback records*, and *update records*. We shall follow SimpleDB and assume two kinds of update record: one for updates to integers, and one for updates to strings.

Three activities that cause log records to be written are:

- *A start record is written when a transaction begins.*
- *A commit or rollback record is written when a transaction completes.*
- *An update record is written when a transaction modifies a value.*

Another potentially loggable activity is appending a block to the end of a file. Then if the transaction rolls back, the new block allocated by *append* could be deallocated from the file. On the other hand, there is no harm in letting the new block remain, because it will not affect other transactions. For simplicity, we shall ignore the possibility of an append log record. Exercise 14.48 addresses the issue.

As an example, consider again the code fragment from Figure 14-3, which incremented the integer at offset 392 of block 3 of file *junk*. Suppose that the transaction's ID is 27, and that the old value at that offset was the integer 542. Figure 14-4 contains the log records that would be generated from this code.

```
<START,  27>
<SETINT, 27, junk, 3, 392, 542, 543>
<COMMIT, 27>
```

Figure 14-4: The log records generated from Figure 14-3

Note that each log record contains a description of what type of record it is (START, SETINT, SETSTRING, COMMIT, or ROLLBACK), and the ID of its transaction. Update records contain five additional things: the name and block number of the modified file, the offset where the modification occurred, the old value at that offset, and the new value at that offset.

In general, multiple transactions will be writing to the log concurrently, and so the log records for a given transaction will be dispersed throughout the log.

14.3.2 Rollback

One use of the log is to help the recovery manager *roll back* a specified transaction.

> *The recovery manager rolls back a transaction*
> *by undoing its modifications.*

Since these modifications are listed in the update log records, it is a relatively simple matter to scan the log, find each update record, and restore the original contents of each modified value. Figure 14-5 presents the algorithm.

```
1. Set the current record to be the most recent log record.
2. Do until the current record is the start record for T:
       a) If the current record is an update record for T then:
              Write the saved old value to the specified location.
       b) Move to the previous record in the log.
3. Append a rollback record to the log.
```

Figure 14-5: The algorithm for rolling back transaction T

Why does this algorithm read the log backwards from the end, instead of forwards from the beginning? There are two reasons. One reason is that the beginning of the log file will contain records from long-ago completed transactions. It is most likely that the records we are looking for are at the end of the log, and thus it is more efficient to read from the end. The second, more important reason is to ensure correctness. Suppose that the value at a location was modified several times. Then there will be several log records for that location, each having a different value. The value to be restored should come from the earliest of these records. If the log records are processed in reverse order, then this will in fact occur.

14.3.3 Recovery

Another use of the log is to *recover* the database.

> *Recovery is performed each time the database system starts up.*
> *Its purpose is to restore the database to a reasonable state.*

The term "reasonable state" means two things:

- All uncompleted transactions should be rolled back.
- All committed transactions should have their modifications written to disk.

When the database system starts up following a normal shutdown, the database should already be in a reasonable state, because the normal shutdown procedure is to wait until the existing transactions complete and then flush all buffers. However, if a crash had caused the system to go down unexpectedly, then there may be uncompleted transactions whose executions were lost. Since there is no way the system can complete them, their modifications must be undone. There may also be committed transactions whose modifications were not yet flushed to disk; these modifications must be redone.

During recovery, modifications made by uncompleted transactions are undone, and modifications made by committed transactions are redone.

The recovery manager assumes that a transaction completed if the log file contains a commit or rollback record for it. So if a transaction had committed prior to the system crash but its commit record did not make it to the log file, then the recovery manager will treat the transaction as if it did not complete. This situation might not seem fair, but there is really nothing else that the recovery manager can do. All it knows is what is contained in the log file, because everything else about the transaction was wiped out in the system crash.

Actually, rolling back a committed transaction is not only unfair, it violates the ACID property of *durability*. Consequently, the recovery manager must ensure that such a scenario cannot occur.

The recovery manager must flush the commit log record to disk before it completes a commit operation.

Recall that flushing a log record also flushes all previous log records. So when the recovery manager finds a commit record in the log, it knows that all of the update records for that transaction are also in the log.

Each update log record contains both the old value and the new value of the modification. The old value is used when we want to undo the modification, and the new value is used when we want to redo the modification. Figure 14-6 presents the recovery algorithm.

```
// The undo stage
1.  For each log record (reading backwards from the end):
        a) If the current record is a commit record then:
                Add that transaction to the list of committed transactions.
        b) If the current record is a rollback record then:
                Add that transaction to the list of rolled-back transactions.
        c) If the current record is an update record and
           that transaction is not on the committed or rollback list, then:
                Restore the old value at the specified location.

// The redo stage
2.  For each log record (reading forwards from the beginning):
        If the current record is an update record and
        that transaction is on the committed list, then:
                Restore the new value at the specified location.
```

Figure 14-6: The undo-redo algorithm for recovering a database

Stage 1 undoes the uncompleted transactions. As with the rollback algorithm, the log must be read backwards from the end to ensure correctness. Reading the log backwards also means that a commit record will always be found before its update records; so when the algorithm encounters an update record, it knows whether that record needs to be undone or not.

Note that stage 1 must read the entire log. For example, the very first transaction might have made a change to the database before going into an infinite loop. That update record will not be found unless we read to the very beginning of the log.

Stage 2 redoes the committed transactions. Since the recovery manager cannot tell which buffers were flushed and which were not, it redoes all changes made by all committed transactions.

The recovery manager performs stage 2 by reading the log forwards from the beginning. The recovery manager knows which update records need to be redone, because it computed the list of committed transaction during stage 1. Note that the log *must* be read forward during the redo stage. If several committed transactions happened to modify the same value, then the final recovered value should be due to the most recent modification.

Note that the recovery algorithm is oblivious to the current state of the database. The recovery manager writes old or new values to the database without looking at what the current values are at those locations, because the log tells it *exactly* what the contents of the database should be. There are two consequences to this feature:

- Recovery is idempotent.
- Recovery may cause more disk writes than necessary.

By *idempotent*, we mean that performing the recovery algorithm several times has the same result as performing it once. In fact, you will still get the same result even if you re-run the recovery algorithm immediately after having run just part of it. This property is essential to the correctness of the algorithm. Suppose, for example, that the database system crashes while it is in the middle of the recovery algorithm. When the database system restarts, it will run the recovery algorithm again, from the beginning. If the algorithm were not idempotent, then re-running it would corrupt the database.

Because this algorithm does not look at the current contents of the database, it may make unnecessary changes. For example, suppose that the modifications made by a committed transaction have been written to disk; then redoing those changes during stage 2 will set the modified values to the contents that they already have. The algorithm can be revised so that it does not make these unnecessary disk writes; see Exercise 14.44.

14.3.4 Undo-only and redo-only recovery

The recovery algorithm of the previous section performs both undo and redo operations. Some database systems choose to simplify the algorithm so that it performs only undo operations or only redo operations; that is, it executes either stage 1 or stage 2 of the algorithm, but not both.

Undo-only recovery

Stage 2 can be omitted if the recovery manager is sure that all committed modifications have been written to disk. The recovery manager can do so by forcing the buffers to disk *before* it writes the commit record to the log. Figure 14-7 expresses this approach as an algorithm. The recovery manager must follow the steps of this algorithm in exactly the order given.

1. Flush the transaction's modified buffers to disk.
2. Write the commit log record.
3. Flush the log page containing that log record.

Figure 14-7: The algorithm for committing a transaction, using undo-only recovery

Let's compare undo-only recovery with undo-redo recovery. Undo-only recovery is faster, because it requires only one pass through the log file, instead of two. The log is also a bit smaller, because update records no longer need to contain the new modified value. On the other hand, the *commit* operation is much slower, because it must flush the modified buffers. If we assume that system crashes are infrequent, then undo-redo recovery wins. Not only do transactions commit faster, but there should be fewer overall disk writes due to the postponed buffer flushes.

Redo-only recovery

Stage 1 can be omitted if the recovery manager is sure that all uncommitted buffers have *not* been written to disk. The recovery manager can do so by having each transaction keep its buffers pinned until the transaction completes – a pinned buffer will not get chosen for replacement, and thus its contents will not get flushed. However, when a transaction is rolled back, its modified buffers will need to be "erased". Figure 14-8 gives the necessary revisions to the rollback algorithm.

```
For each buffer modified by the transaction:
    a) Mark the buffer as unallocated. (In SimpleDB, set its block number to -1)
    b) Mark the buffer as unmodified.
    c) Unpin the buffer.
```

Figure 14-8: The algorithm for rolling back a transaction, using redo-only recovery

Redo-only recovery is faster than undo-redo recovery, because uncommitted transactions can be ignored. However, it requires that each transaction keep a buffer pinned for every block that it modifies, which increases the contention for buffers in the system. In a large database, this contention can seriously impact the performance of all transactions, which makes redo-only recovery a risky choice.

It is interesting to think about whether it is possible to combine the undo-only and redo-only techniques, to create a recovery algorithm that doesn't require either stage 1 or stage 2. See Exercise 14.19.

14.3.5 Write-ahead logging

We need to examine step 1 of the recovery algorithm in more detail. Recall that this step iterates through the log, performing an undo for each update record from an uncompleted transaction. In justifying the correctness of this step, we made the following assumption: *all update records for an uncompleted transaction will be in the log file.* An update log record that doesn't make it to disk cannot be undone, which means that the database would become corrupted.

Since the system could crash at any time, the only way to satisfy this assumption is to have the log manager flush each update record to disk as soon as it is written. But as we saw in Section 13.2, this strategy is painfully inefficient. We need a better way.

Let's analyze the kinds of things that can go wrong. Suppose that an uncompleted transaction modified a page, causing an update log record to be created. If the server crashes, there are four possibilities:

a) Both the page and the log record got written to disk.
b) Only the page got written to disk.
c) Only the log record got written to disk.
d) Neither got written to disk.

Consider each possibility in turn. If (a), then the recovery algorithm will find the log record and undo the change to the data block on disk; no problem. If (b), then the recovery algorithm won't find the log record, and so it will not undo the change to the data block. This is a serious problem. If (c), then the recovery algorithm will find the log record, and will undo the non-existent change to the block. Since the block wasn't actually changed, this is a waste of time, but not incorrect. If (d), then the recovery algorithm won't find the log record, but since there was no change to the block, there is nothing to undo anyway; no problem.

Thus (b) is the only problem case. This case can be avoided if the recovery manager ensures that an update log record always gets written to disk before the corresponding modified buffer page. This strategy is called using a *write-ahead log*. Note that the log may describe modifications to the database that never wind up occurring (as in possibility (c) above), but if the database does get modified, the modification will always be in the log.

> *In the write-ahead log strategy, a modified buffer can be written to disk only after all of its update log records have been written to disk.*

The standard way to implement a write-ahead log is to have each buffer save the LSN of the log record corresponding to its most recent modification. Before a buffer replaces a modified page, it tells the log manager to flush the log up to its LSN. Log records having a higher LSN are not affected, and will not get flushed.

14.3.6 Quiescent checkpointing

静态检查点

The log contains the history of every modification to the database. As time passes, the size of the log file can become very large – in some cases, larger than the data files. The prospect of having to read the entire log during recovery and undo/redo every change to the database is daunting. Consequently, recovery strategies have been devised for reading only a portion of the log. The basic idea is the following.

> *The recovery algorithm can stop searching the log as soon as it knows:*
> - *all earlier log records were written by completed transactions; and*
> - *the buffers for those transactions have been flushed to disk.*

The first bullet point applies to the undo stage of the recovery algorithm. It ensures that there are no more uncommitted transactions to be rolled back. The second bullet point applies to the redo stage, and ensures that all earlier committed transactions do not need to be redone. Note that if the recovery manager implements undo-only recovery, then the second bullet point will always be true.

At any point in time, the recovery manager can perform a *quiescent checkpoint* operation, as shown in Figure 14-9. Step 2 of that algorithm ensures that the first bullet point is satisfied, and step 3 ensures that the second bullet point is satisfied.

1. Stop accepting new transactions.
2. Wait for existing transactions to finish.
3. Flush all modified buffers.
4. Append a quiescent checkpoint record to the log and flush it to disk.
5. Start accepting new transactions.

Figure 14-9: The algorithm for performing a quiescent checkpoint

The quiescent checkpoint record acts as a marker in the log. When stage 1 of the recovery algorithm encounters the checkpoint record as it moves backwards through the log, it knows that all earlier log records can be ignored; it therefore can begin stage 2 from that point in the log and move forward.

The recovery algorithm never needs to look at the log records prior to a quiescent checkpoint record.

A good time to write a quiescent checkpoint record is during system startup, after recovery has completed and before new transactions have begun. Since the recovery algorithm has just finished processing the log, the recovery manager will never need to examine those log records again.

For an example, consider the log shown in Figure 14-10. This example log illustrates three things: First, no new transactions can start once the checkpoint process begins; second, the checkpoint record was written as soon as the last transaction completed and the buffers were flushed; and third, other transactions may start as soon as the checkpoint record is written.

```
<START, 0>
<SETINT, 0, junk, 33, 8, 542, 543>
<START, 1>
<START, 2>
<COMMIT, 1>
<SETSTRING, 2, junk, 44, 20, hello, ciao>
            //The quiescent checkpoint procedure starts here
<SETSTRING, 0, junk, 33, 12, joe, joseph>
<COMMIT, 0>
            //tx 3 wants to start here, but must wait
<SETINT, 2, junk, 66, 8, 0, 116>
<COMMIT, 2>
<CHECKPOINT>
<START, 3>
<SETINT, 3, junk, 33, 8, 543, 120>
```

Figure 14-10: A log using quiescent checkpointing

14.3.7 Nonquiescent checkpointing

Quiescent checkpointing is simple to implement and easy to understand. However, it requires that the system be unavailable while the recovery manager waits for existing transactions to complete. In many database applications this is a serious shortcoming – Companies don't want their databases to occasionally stop responding for arbitrary periods of time. Consequently, a checkpointing algorithm has been developed that doesn't require quiescence. The algorithm appears in Figure 14-11.

1. Let $T_1,\ldots,T_k$ be the currently running transactions.
2. Stop accepting new transactions.
3. Flush all modified buffers.
4. Write the record `<NQCKPT` T_1`,…,`T_k`>` into the log.
5. Start accepting new transactions.

Figure 14-11: The algorithm for adding a nonquiescent checkpoint record

This algorithm uses a different kind of checkpoint record, called a *nonquiescent checkpoint record*. A non-quiescent checkpoint record contains a list of the currently-running transactions.

We can revise the recovery algorithm as follows. Stage 1 of the algorithm reads the log backwards as before, and keeps track of the completed transactions. If it encounters a nonquiescent checkpoint record `<NQCKPT` T_1`,…,`T_k`>`, it determines which of these T_i transactions are still running. It can then continue reading the log backwards until it encounters the start record for the earliest of those transactions. All log records prior to this start record can be ignored.

> *The recovery algorithm never needs to look at the log records prior to the start record of the earliest transaction listed in a nonquiescent checkpoint record.*

For an example, consider again the log of Figure 14-10. Using nonquiescent checkpointing, it would appear as in Figure 14-12. Note that the `<NQCKPT…>` record appears in this log in the place where the checkpoint process began in Figure 14-10, and states that transactions 0 and 2 are still running at that point. This log differs from that of Figure 14-10 in that transaction 2 never commits.

```
<START, 0>
<SETINT, 0, junk, 33, 8, 542, 543>
<START, 1>
<START, 2>
<COMMIT, 1>
<SETSTRING, 2, junk, 44, 20, hello, ciao>
<NQCKPT, 0, 2>
<SETSTRING, 0, junk, 33, 12, joe, joseph>
<COMMIT, 0>
<START, 3>
<SETINT, 2, junk, 66, 8, 0, 116>
<SETINT, 3, junk, 33, 8, 543, 120>
```

Figure 14-12: A log using nonquiescent checkpointing

If the recovery algorithm sees this log during system startup, it would enter stage 1 and proceed as follows.

- When it encounters the `<SETINT, 3, ...>` log record, it will check to see if transaction 3 was on the list of committed transactions. Since that list is currently empty, the algorithm will perform an undo, writing the integer 543 to offset 8 of block 33.
- The log record `<SETINT, 2, ...>` will be treated similarly, writing the integer 0 to offset 8 of block 66.
- The `<COMMIT, 0>` log record will cause 0 to be added to the list of committed transactions.
- The `<SETSTRING, 0, ...>` log record will be ignored, because 0 is in the committed transaction list.
- When it encounters the `<NQCKPT 0,2>` log record, it knows that transaction 0 has committed, and thus it can ignore all log records prior to the start record for transaction 2.
- When it encounters the `<START, 2>` log record, it enters stage 2 and begins moving forward through the log.
- The `<SETSTRING, 0, ...>` log record will be redone, because 0 is in the committed transaction list. The value 'joseph' will be written to offset 12 of block 33.

14.3.8 Data item granularity

The recovery management algorithms of this section log and restore *values*. That is, an update log record is created each time a value is modified, with the log record containing the previous and new versions of the value. We call this unit of logging a *recovery data item*.

> *A recovery data item is the unit of logging used by the recovery manager.*
> *The size of a data item is called its granularity.*

Instead of using values as data items, the recovery manager could choose to use blocks or files. For example, suppose that blocks were chosen as the data item. In this case, an update log record would be created each time a block was modified, with the previous and new values of the block being stored in the log record.

The advantage to logging blocks is that fewer log records are needed if we use undo-only recovery. Suppose that a transaction pins a block, modifies several values, and then unpins it. We could save the original contents of the block in a single log record, instead of having to write one log record for each modified value. The disadvantage, of course, is that the update log records are now very large; the entire contents of the block gets saved, regardless of how many of its values actually change. Thus, logging blocks makes sense only if transactions tend to do a lot of modifications per block.

Now consider what it would mean to use files as data items. A transaction would generate one update log record for each file that it changed. Each log record would contain the entire original contents of that file. To roll back a transaction, we would just need to replace the existing files with their original versions. This approach is almost certainly less practical than using values or blocks as data items, because each transaction would have to make a copy of the entire file, no matter how many values changed.

Although file-granularity data items are impractical for database systems, they are often used by non-database applications. Suppose for example that the system crashes while you are editing a file. After the system reboots, some word processors are able to show you two versions of the file: the version that you most recently saved, and the version that existed at the time of the crash. The reason is that those word processors do not write your modifications directly to the original file, but to a copy; when you save, the modified file is copied to the original one. This strategy is a crude version of file-based logging.

14.3.9 The SimpleDB recovery manager

The SimpleDB recovery manager is implemented via the package *simpledb.tx.recovery*, and in particular the public class *RecoveryMgr*. The API for class *RecoveryMgr* appears in Figure 14-13.

RecoveryMgr

```
public RecoveryMgr(int txnum);
public void commit();
public void rollback();
public void recover();
public int  setInt(Buffer buff, int offset, int newval);
public int  setString(Buffer buff, int offset, String newval);
```

Figure 14-13: The API for the SimpleDB recovery manager

Each transaction creates its own recovery manager. The recovery manager has methods to write the appropriate log records for that transaction. For example, the constructor writes a start log record to the log; the *commit* and *rollback* methods write corresponding log records; and the *setInt* and *setString* methods extract the old value from the specified buffer and write an update record to the log. The *rollback* and *recover* methods also perform the rollback (or recovery) algorithms.

The SimpleDB code makes the following design decisions:

> *The SimpleDB recovery manager chooses to use*
> *undo-only recovery with value-granularity data items.*

The code for the SimpleDB recovery manager can be divided into three areas of concern:

- code to implement log records;
- code to iterate through the log file;
- code to implement the rollback and recovery algorithms.

Log Records

A log record is implemented as an array of values. The first value in the record is an integer that denotes the *operator* of the record; the operator can be one of CHECKPOINT, START, COMMIT, ROLLBACK, SETINT, or SETSTRING. The remaining values in the record depend on the operator – a quiescent checkpoint record has no other values, an update record has five other values, and the other records have one other value. Each kind of log record has it own class that implements the interface *LogRecord*; see Figure 14-14.

```
public interface LogRecord {
   static final int CHECKPOINT = 0, START = 1,
                    COMMIT = 2, ROLLBACK  = 3,
                    SETINT = 4, SETSTRING = 5;

   static final LogMgr logMgr = SimpleDB.logMgr();

   int writeToLog();
   int op();
   int txNumber();
   void undo(int txnum);
}
```

Figure 14-14: The code for the SimpleDB *LogRecord* interface

The interface defines a method *writeToLog*, which appends the record to the log and returns its LSN. The interface also defines three methods that extract the components of the log record. Method *op* returns the record's operator. Method *txNumber* returns the ID of the transaction that wrote the log record. This method makes sense for all log records except checkpoint records, which return a dummy ID value. And the method *undo* restores any changes stored in that record. Only the *setint* and *setstring* log records will have a non-empty *undo* method; the method for those records will pin a buffer to the specified block, write the specified value at the specified offset, and unpin the buffer.

The classes for the individual kinds of log record all have similar code; it should suffice to examine the code for one of the classes, say *SetStringRecord*; see Figure 14-15.

```
public class SetStringRecord implements LogRecord {
   private int mytxnum, offset;
   private String val;
   private Block blk;

   public SetStringRecord(int txnum, Block blk, int offset, String val) {
      this.txnum = txnum;
      this.blk = blk;
      this.offset = offset;
      this.val = val;
   }

   public SetStringRecord(BasicLogRecord rec) {
      mytxnum = rec.nextInt();
      String filename = rec.nextString();
      int blknum = rec.nextInt();
      blk = new Block(filename, blknum);
      offset = rec.nextInt();
      val = rec.nextString();
   }

   public int writeToLog() {
      Object[] rec = new Object[] {SETSTRING, txnum, blk.fileName(),
                                   blk.number(), offset, val};
      return logMgr.append(rec);
   }

   public int op() {
      return LogRecord.SETSTRING;
   }

   public int txNumber() {
      return mytxnum;
   }

   public String toString() {
      return "<SETSTRING " + mytxnum + " " + blk + " " + offset
              + " " + val + ">";
   }

   public void undo(int txnum) {
      BufferMgr buffMgr = SimpleDB.bufferMgr();
      Buffer buff = buffMgr.pin(blk);
      buff.setString(offset, val, txnum, -1);
      buffMgr.unpin(buff);
   }
}
```

Figure 14-15: The code for the class *SetStringRecord*

The class has two constructors: One constructor has an argument for each component of the record, and simply stores them in the object; this constructor is called right before the record is written to the log. The other constructor is called by the *rollback* and *recover* methods. They pass a *BasicLogRecord* object into the constructor, which then reads the appropriate values from it. The *undo* method pins a buffer to the modified block, restores the saved value, and then unpins the block.

One minor glitch to the implementation of *undo* is that the *setString* method expects to get the LSN of the log record corresponding to the modification. In this case, of course, there is no point in logging the restored value, and so *undo* passes a dummy LSN to *setString*.

<u>Iterating Through the Log</u>

The class *LogRecordIterator* allows a client to iterate backwards through the log file, one record at a time; see Figure 14-16.

```
class LogRecordIterator implements Iterator<LogRecord> {
  private Iterator<BasicLogRecord> iter = SimpleDB.logMgr().iterator();

   public boolean hasNext() {
      return iter.hasNext();
   }

   public LogRecord next() {
      BasicLogRecord rec = iter.next();
      int op = rec.nextInt();
      switch (op) {
         case CHECKPOINT:
            return new CheckpointRecord(rec);
         case START:
            return new StartRecord(rec);
         case COMMIT:
            return new CommitRecord(rec);
         case ROLLBACK:
            return new RollbackRecord(rec);
         case SETINT:
            return new SetIntRecord(rec);
         case SETSTRING:
            return new SetStringRecord(rec);
         default:
            return null;
      }
   }
}
```

Figure 14-16: The code for the class *LogRecordIterator*

Its code is straightforward. Its constructor calls the log manager to obtain an iterator of basic log records. The method *next* obtains the next basic log record from the iterator, and reads an integer from it. This integer will be the operator of the next log record. The method uses this value to determine which class constructor to call, and passes the basic log record to that constructor so that the remaining values can be read.

As an example, the following code prints the contents of the log in reverse order.

```
Iterator<LogRecord> iter = new LogRecordIterator();
while (iter.hasNext()) {
   LogRecord rec = iter.next();
   System.out.println(rec.toString());
}
```

Rollback and Recover

The class *RecoveryMgr* implements the recovery manager API; its code appears in Figure 14-17. The code for each method implements the undo-only algorithm discussed earlier. In particular, the *commit* and *rollback* methods flush the transaction's buffers before writing their log record; and the *doRollback* and *doRecover* methods make a single backward pass through the log.

```
public class RecoveryMgr {
   private int txnum;

   public RecoveryMgr(int txnum) {
      this.txnum = txnum;
      new StartRecord(txnum).writeToLog();
   }

   public void commit() {
      SimpleDB.bufferMgr().flushAll(txnum);
      int lsn = new CommitRecord(txnum).writeToLog();
      SimpleDB.logMgr().flush(lsn);
   }

   public void rollback() {
      SimpleDB.bufferMgr().flushAll(txnum);
      doRollback();
      int lsn = new RollbackRecord(txnum).writeToLog();
      SimpleDB.logMgr().flush(lsn);
   }

   public void recover() {
      SimpleDB.bufferMgr().flushAll(txnum);
      doRecover();
      int lsn = new CheckpointRecord().writeToLog();
      SimpleDB.logMgr().flush(lsn);
   }

   public int setInt(Buffer buff, int offset, int newval) {
      int oldval = buff.getInt(offset);
      Block blk = buff.block();
      if (isTemporaryBlock(blk))
         return -1;
      else
         return new SetIntRecord(txnum, blk, offset,
                                 oldval).writeToLog();
   }

   public int setString(Buffer buff, int offset, String newval) {
      String oldval = buff.getString(offset);
      Block blk = buff.block();
      if (isTemporaryBlock(blk))
         return -1;
      else
         return new SetStringRecord(txnum, blk, offset,
                                    oldval).writeToLog();
   }

   private void doRollback() {
      Iterator<LogRecord> iter = new LogRecordIterator();
      while (iter.hasNext()) {
         LogRecord rec = iter.next();
         if (rec.txNumber() == txnum) {
            if (rec.op() == START)
               return;
            rec.undo(txnum);
         }
      }
   }

   private void doRecover() {
      Collection<Integer> committedTxs = new ArrayList<Integer>();
      Iterator<LogRecord> iter = new LogRecordIterator();
      while (iter.hasNext()) {
         LogRecord rec = iter.next();
         if (rec.op() == CHECKPOINT)
            return;
         if (rec.op() == COMMIT)
            committedTxs.add(rec.txNumber());
         else if (!committedTxs.contains(rec.txNumber()))
            rec.undo(txnum);
      }
   }

   private boolean isTemporaryBlock(Block blk) {
      return blk.fileName().startsWith("temp");
}
```

Figure 14-17: The code for the class *RecoveryMgr*

The *doRollback* method reads log records until it hits the start record for the given transaction. Each time it finds a log record for that transaction, it calls the record's *undo* method. It stops when it encounters the start record for that transaction.

The *doRecover* method is implemented similarly. It reads the log until it hits a quiescent checkpoint record or reaches the end of the log, keeping a list of committed transaction numbers. It undoes uncommitted update records the same as in *rollback*, the difference being that it processes the records for any uncommitted transaction, not just for a specific one. This method is slightly different from the recovery algorithm of Figure 14-6, because it will undo transactions that have already been rolled back. Although this difference does not make the code incorrect, it does make it less efficient. Exercise 14.44 asks you to improve it.

One further detail in the code is that the *setInt* and *setString* methods do not log updates to temporary files – that is, files beginning with "temp". Temporary files are created during materialized query processing (see Chapter 22) and are deleted afterwards, so there is no point in recovering their changes.

14.4 Concurrency Management

The *concurrency manager* is the portion of the server that is responsible for the correct execution of concurrent transactions. In this section we will examine what it means for execution to be "correct", and study the algorithms that are used to ensure correctness.

14.4.1 Serializable schedules

We define the *history* of a transaction to be the sequence of calls to the methods that access a database file – namely, the *get/set* methods[†]. For example, look back at Figure 14-3, and consider the first two parts of transaction *tx*. Its history could be written, somewhat tediously, as follows:

```
int    n = tx.getInt(new Block("junk", 3), 392);
String s = tx.getString(new Block("junk", 3), 20);
tx.setInt(new Block("junk", 3), 392, n+1);
```

A less tedious way to express the history of a transaction is in terms of the affected blocks:

```
tx:  R(junk,3); R(junk,3); W(junk,3);
```

That is, this history states that transaction *tx* reads twice from block 3 of file *junk* and then writes to block 3 of file *junk*.

> *The history of a transaction describes the sequence of database actions made by that transaction.*

The term "database action" is deliberately vague. The above examples treated a database action first as the modification of a value, and then as the read/write of a disk block. Other granularities are possible, which are discussed in Section 14.4.8. Until then, we shall assume that a database action is the reading or writing of a disk block.

When multiple transactions are running concurrently, the database system will interleave the execution of their threads, periodically interrupting one thread and resuming another.

[†] The *size* and *append* methods also access a database file, but more subtly than do the get/set methods. We will consider the effect of *size* and *append* later, in Section 14.4.5.

(In SimpleDB, the Java runtime environment does this automatically.) Thus the actual sequence of operations performed by the concurrency manager will be an unpredictable interleaving of the histories of its transactions. We call that interleaving a *schedule*.

A schedule is the interleaving of the histories of the transactions, as executed by the database system.

The purpose of concurrency control is to ensure that only correct schedules get executed. But what should we mean by "correct"? Well, consider the simplest possible schedule – one in which all transactions run serially. The operations in this schedule will not be interleaved; that is, the schedule will simply be the back-to-back histories of each transaction. We call this kind of schedule a *serial schedule*.

A serial schedule is a schedule in which the histories are not interleaved.

Concurrency control is predicated on the assumption that a serial schedule *has* to be correct, since there is no concurrency.

The interesting thing about using serial schedules to define correctness is that different serial schedules of the same transactions can give different results. For example, consider the two transactions T_1 and T_2, having the following identical histories:

T_1: W(b_1); W(b_2);
T_2: W(b_1); W(b_2);

Although these transactions have the same history (i.e., they both write block b_1 first and then block b_2), they are not necessarily identical as transactions – for example, T_1 might write an 'X' at the beginning of each block, whereas T_2 might write a 'Y'. If T_1 executes before T_2, the blocks will contain the values written by T_2; but if they execute in the reverse order, the blocks will contain the values written by T_1.

In this example, T_1 and T_2 have different opinions about what blocks b_1 and b_2 should contain. And, since all transactions are equal in the eyes of the database system, there is no way to say that one result is more correct than another. Thus we are forced to admit that the result of *either* serial schedule is correct. That is, there can be several correct results.

Given a non-serial schedule, we say that it is *serializable* if it produces the same result as some serial schedule[†]. Since serial schedules are correct, it follows that serializable schedules must also be correct.

A serializable schedule is a schedule that produces the same result as some serial schedule.

[†] The term *serializable* is also used in Java – a serializable class is one whose objects can be written as a stream of bytes. Unfortunately, that use of the term has absolutely nothing to do with our use of it.

Consider the following non-serial schedule of the above example transactions:

$$W_1(b_1);\ W_2(b_1);\ W_1(b_2);\ W_2(b_2);$$

Here we use $W_1(b_1)$ to mean that transaction T_1 writes block b_1, etc. This schedule results from running the first half of T_1, followed by the first half of T_2, the second half of T_1, and the second half of T_2. This schedule is serializable, because it is equivalent to doing T_1 first, then T_2. On the other hand, consider the following schedule:

$$W_1(b_1);\ W_2(b_1);\ W_2(b_2);\ W_1(b_2);$$

This transaction does the first half of T_1, all of T_2, and then the second half of T_1. The result of this schedule is that block b_1 contains the values written by T_2, but block b_2 contains the values written by T_1. This result cannot be produced by any serial schedule, and so we say that the schedule is *non-serializable*.

Recall the ACID property of *isolation*, which said that each transaction should execute as if it were the only transaction in the system. A non-serializable schedule does not have this property. Therefore, we are forced to admit that non-serializable schedules are incorrect. This analysis can be expressed as the following principle:

> *A schedule is correct if and only if it is serializable.*

14.4.2 The lock table

The database system is responsible for ensuring that all schedules are serializable. A common technique is to use *locking* to postpone the execution of a transaction. In Section 14.4.3 we will look at how locking can be used to ensure serializability. In this section, we simply examine how the basic locking mechanism works.

Each block has two kinds of lock – a *shared* lock (or *slock*) and an *exclusive* lock (or *xlock*). If a transaction has an exclusive lock on a block, then no other transaction is allowed to have any kind of lock on it; if the transaction has a shared lock on a block, then other transactions are only allowed to have shared locks on it.

The *lock table* is the portion of the server that is responsible for granting locks to transactions. In SimpleDB, the lock table is implemented via the class *LockTable*. Its API is given in Figure 14-18.

```
public void sLock(Block blk);
public void xLock(Block blk);
public void unlock(Block blk);
```

Figure 14-18: The API for the SimpleDB class *LockTable*

The method *sLock* requests a shared lock on the specified block. If an exclusive lock already exists on the block, the method waits until the exclusive lock has been released. The method *xLock* requests an exclusive lock on the block. This method waits until no other transaction has any kind of lock on it. The *unlock* method releases a lock on the block.

Figure 14-19 presents an example demonstrating the interactions between lock requests.

Three threads wish to lock blocks b_1 and b_2 as follows:

```
Thread A:  sLock(b1);  sLock(b2);  unlock(b1);  unlock(b2);
Thread B:  xLock(b2);  sLock(b1);  unlock(b1);  unlock(b2);
Thread C:  xLock(b1);  sLock(b2);  unlock(b1);  unlock(b2);
```

Suppose that the threads happen to execute so that they are interrupted after each statement. Then:

1. Thread A obtains an slock on b_1.
2. Thread B obtains an xlock on b_2.
3. Thread C cannot get an xlock on b_1, because someone else has a lock on it. Thus thread C waits.
4. Thread A cannot get an slock on b_2, because someone else has an xlock on it. Thus thread A also waits.
5. Thread B can continue. It obtains an slock on block b_1, because nobody currently has an xlock on it. (It doesn’t matter that thread C is waiting for an xlock that block.)
6. Thread B unlocks block b_1, but this does not help either waiting thread.
7. Thread B unlocks block b_2.
8. Thread A can now continue, and obtains its slock on b_2.
9. Thread A unlocks block b_1.
10. Thread C finally is able to obtain its xlock on b_1.
11. Threads A and C can continue in any order until they complete.

Figure 14-19: An example of the interaction between lock requests

14.4.3 The lock protocol

We now tackle the question of how locking can be used to ensure that all schedules are serializable.

Consider two transactions having the following histories:

```
T1: R(b1); W(b2);
T2: W(b1); W(b2);
```

What is it that causes their serial schedules to have different results? Transactions T_1 and T_2 both write to the same block b_2, which means that the order of these operations makes a difference – whichever transaction writes last is the "winner". We say that their operations $\{W_1(b_2), W_2(b_2)\}$ *conflict*. In general, two operations conflict if the order in which they are executed can produce a different result. If two transactions have conflicting operations, then their serial schedules may have different (but equally correct) results.

The above two conflicts are examples of *write-write* conflicts. A second kind of conflict is a *read-write* conflict. For example, the operations $\{R_1(b_1), W_2(b_1)\}$ conflict – if $R_1(b_1)$ executes first, then T_1 reads one version of block b_1, whereas if $W_2(b_1)$ executes first, then T_1 reads a different version of block b_1. Note that two read operations cannot ever conflict, nor can operations involving different blocks.

The reason that we care about conflicts is because they influence the serializablility of a schedule.

The order in which conflicting operations are executed in a non-serial schedule determines the equivalent serial schedule.

In the above example, if $W_2(b_1)$ executes before $R_1(b_1)$, then any equivalent serial schedule must have T_2 running before T_1. This means that if you consider all operations in T_1 that conflict with T_2, then either they all must be executed before any conflicting T_2 operations, or they all must be executed after them. Non-conflicting operations can happen in an arbitrary order[†].

Locking can be used to avoid write-write and read-write conflicts. In particular, suppose we require that all transactions use locks according to the protocol of Figure 14-20.

1. Before reading a block, acquire a shared lock on it.
2. Before modifying a block, acquire an exclusive lock on it.
3. Release all locks after a commit or rollback.

Figure 14-20: The lock protocol

From this protocol we can infer two important facts. First, if a transaction gets a shared lock on a block, then no other active transaction will have written to the block (otherwise, some transaction would still have an exclusive lock on the block). Second, if a transaction gets an exclusive lock on a block, then no other active transaction will have accessed the block in any way (otherwise, some transaction would still have a lock on the block). These facts imply that an operation performed by a transaction will never conflict with a previous operation by another active transaction.

[†] Actually, you can construct obscure examples in which certain write-write conflicts can also occur in any order; see Exercise 14.26. Such examples, however, are not practical enough to be worth considering.

In other words, we have shown the following fact.

> *If all transactions to obey the lock protocol, then:*
> - *the resulting schedule will always be serializable (and hence correct);*
> - *the equivalent serial schedule is determined by the order in which the transactions commit.*

The lock protocol forces transactions to hold their locks until they complete, which can drastically limit the concurrency in the system. It would be nice if a transaction could release its locks when they are no longer needed, so other transactions wouldn't have to wait as long. However, two serious problems can arise if a transaction releases its locks before it completes.

> *Two problems can occur when a transaction releases its locks early:*
> - *It may no longer be serializable.*
> - *Other transactions can read its uncommitted changes.*

Serializability Problems

Once a transaction unlocks a block, it can no longer lock another block without impacting serializability. To see why, consider a transaction T_1 that unlocks block x before locking block y.

```
T1:  ... R(x); UL(x);  SL(y); R(y); ...
```

Suppose that T_1 is interrupted during the time interval between the unlock of x and the slock of y. At this point T_1 is extremely vulnerable, because both x and y are unlocked. Suppose that another transaction T_2 jumps in, locks both x and y, writes to them, commits, and releases its locks. We then have the following situation: T_1 must come before T_2 in the serial order, because T_1 read the version of block x that existed before T_2 wrote it. On the other hand, T_1 must also come after T_2 in the serial order, because T_1 will read the version of block y written by T_2. Thus the resulting schedule is non-serializable.

It can be shown that the converse is also true – that is, if a transaction acquires all of its locks before unlocking any of them, the resulting schedule is guaranteed to be serializable. (See Exercise 14.27.) This variant of the lock protocol is called *two-phase locking*. This name comes from the fact that under this protocol, a transaction has two phases – the phase where it accumulates the locks, and the phase where it releases the locks.

> *The two-phase lock protocol allows a transaction to begin releasing locks as soon as it has acquired all of them.*

Although two-phase locking is theoretically a more general protocol, a database system cannot easily take advantage of it. Usually by the time a transaction has finished accessing its final block (which is when locks are finally able to be released), it is ready to commit anyway. So the fully-general two-phase locking protocol is rarely effective in practice.

Reading Uncommitted Data

Another problem with releasing locks early is that transactions will be able to read uncommitted data. Consider the following partial schedule:

$$\ldots\ W_1(b);\ UL_1(b);\ SL_2(b);\ R_2(b);\ \ldots$$

In this schedule, T_1 writes to block b and unlocks it; transaction T_2 then locks and reads b. If T_1 eventually commits, then there is no problem. But suppose that T_1 does a rollback. Then T_2 will also have to rollback, because its execution is based on changes that no longer exist. And if T_2 rolls back, this could cause still other transactions to roll back. This phenomenon is known as *cascading rollback.*

When the database system lets a transaction read uncommitted data, it enables more concurrency, but it takes the risk that the transaction that wrote the data will not commit. Certainly rollbacks tend to be infrequent, and cascaded rollbacks should be more so. The question is whether a database system wants to take *any* risk of possibly rolling back a transaction unnecessarily. Most commercial database systems are not willing to take this risk, and therefore always wait until a transaction completes before releasing its exclusive locks.

14.4.4 Deadlock

Although the lock protocol guarantees that schedules will be serializable, it does not guarantee that all transactions will commit. In particular, it is possible for transactions to be *deadlocked.*

We saw an example of deadlock in Section 12.4.3, where two client threads were each waiting for the other to release a buffer. A similar possibility exists with locks. A deadlock occurs when there is a cycle of transactions in which the first transaction is waiting for a lock held by the second transaction, the second transaction is waiting for a lock held by the third transaction, and so on, until the last transaction is waiting for a lock held by the first transaction. In such a case, none of the waiting transactions can continue, and all will wait potentially forever. For a simple example, consider the following histories, in which the transactions write to the same blocks but in different orders:

$$T_1:\ W(b_1);\ W(b_2);$$
$$T_2:\ W(b_2);\ W(b_1);$$

Suppose that T_1 first acquires its lock on block b_1. There is now a race for the lock on block b_2. If T_1 gets it first, then T_2 will wait, T_1 will eventually commit and release its locks, and T_2 can continue. No problem. But if T_2 gets the lock on block b_2 first, then deadlock occurs – T_1 is waiting for T_2 to unlock block b_2, and T_2 is waiting for T_1 to unlock block b_1. Neither transaction can continue.

The concurrency manager can detect a deadlock by keeping a "waits-for" graph. This graph has one node for each transaction, and an edge from T_1 to T_2 if T_1 is waiting for a lock that T_2 holds; each edge is labeled by the block that the transaction is waiting for. Every time a lock is requested or released, the graph is updated. For example, the waits-for graph corresponding to the above deadlock scenario appears in Figure 14-21.

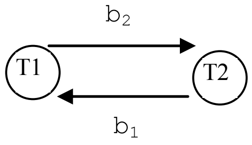


Figure 14-21: A waits-for graph

It is easy to show that deadlock exists iff the waits-for graph has a cycle; see Exercise 14.28. When the transaction manager detects the occurrence of a deadlock, it can break it by summarily rolling back any one of the transactions in the cycle. A reasonable strategy is to roll back the transaction whose lock request "caused" the cycle, although other strategies are possible; see Exercise 14.29.

If we consider threads waiting for buffers as well as those waiting for locks, then testing for deadlock gets considerably more complicated. For example, suppose that the buffer pool contains only two buffers, and consider the following scenario:

```
T1:  xlock(b1); pin(b4);
T2:  pin(b2); pin(b3); xlock(b1);
```

Suppose transaction T_1 gets interrupted after obtaining the lock on block b_1, and then T_2 pins blocks b_2 and b_3. T_2 will wind up on the waiting list for xlock(b_1), and T_1 will wind up on the waiting list for a buffer. There is deadlock, even though the waits-for graph is acyclic.

In order to detect deadlock in such situations, the lock manager must not only keep a waits-for graph; it also needs to know about which transactions are waiting for what buffers. Incorporating this additional consideration into the deadlock detection algorithm turns out to be fairly difficult. Adventurous readers are encouraged to try Exercise 14.38.

The problem with using a waits-for graph to detect deadlock is that the graph is somewhat difficult to maintain, and the process of detecting cycles in the graph is time-consuming. Consequently, simpler strategies have been developed to approximate deadlock detection. These strategies are conservative, in the sense that they will always detect a deadlock, but they might also treat a non-deadlock situation as a deadlock. We will consider two such possible strategies; Exercise 14.33 considers another.

The first approximation strategy is called *wait-die*; see Figure 14-22.

Suppose T_1 requests a lock that conflicts with a lock held by T_2.

- T_1 waits for the lock if it is older than T_2.
- T_1 is rolled back (i.e. it "dies") if it is newer than T_2.

Figure 14-22: The wait-die deadlock detection strategy

This strategy ensures that all deadlocks are detected, because the waits-for graph will contain only edges from older transactions to newer transactions. But the strategy also treats every potential deadlock as a cause for rollback. For example, suppose transaction T_1 is older than T_2, and T_2 requests a lock currently held by T_1. Even though this request may not immediately cause deadlock, there is the potential for it, because at some later point T_1 might request a lock held by T_2. Thus the strategy will preemptively roll back T_2.

The second approximation strategy is to use a time limit to detect a possible deadlock. If a transaction has been waiting for some preset amount of time, the transaction manager can assume that it is deadlocked, and will roll it back. See Figure 14-23.

Suppose T_1 requests a lock that conflicts with a lock held by T_2.

1. T_1 waits for the lock.
2. T_1 is rolled back if it stays on the wait list too long.

Figure 14-23: The time-limit deadlock detection strategy

Regardless of the deadlock detection strategy, the concurrency manager must break the deadlock by rolling back an active transaction. The hope is that by releasing that transaction's locks, the remaining transactions will be able to complete. Once the transaction is rolled back, the concurrency manager throws an exception; in SimpleDB, this exception is called a *LockAbortException*. As with the *BufferAbortException* of Chapter 13, this exception is caught by the JDBC client of the aborted transaction, which then decides how to handle it. For example, the client could choose to simply exit, or it could try to run the transaction again.

14.4.5 File-level conflicts and phantoms

We have so far considered the conflicts that arise from the reading and writing of blocks. Another kind of conflict involves the methods *size* and *append*, which read and write the end-of-file marker. These two methods clearly conflict with each other: Suppose that transaction T_1 first calls *append* and then transaction T_2 calls *size*; then T_1 must come before T_2 in any serial order.

One of the consequences of this conflict is known as the *phantom problem*. Suppose that T_2 must read the entire contents of a file multiple times[†], and that it calls *size* before each iteration to determine how many blocks to read. Moreover, suppose that after T_2 reads the file the first time, transaction T_1 appends some additional values to the file and commits. The next time through the file, T_2 will see these additional values, in violation of the ACID property of isolation. These additional values are called *phantoms*, because to T_2 they have shown up mysteriously.

How can the concurrency manager avoid this conflict? The approach that we have seen for avoiding read-write conflicts is to have T_2 obtain an slock on each block that it reads; this way, T_1 will not be able to write new values to those blocks. However, this approach will not work here, because it would require T_2 to slock the new blocks *before* T_1 creates them!

The solution is to allow transactions to lock the end-of-file marker.

The concurrency manager can avoid the phantom problem
if it allows transactions to lock the end-of-file marker.

In particular, a transaction needs to xlock the marker in order to call the *append* method, and it needs to slock the marker in order to call the *size* method. In our above scenario, if T_1 calls *append* first, then T_2 will not be able to determine the file size until T_1 completes; conversely, if T_2 has already determined the file size, then T_1 would be blocked from appending until T_2 commits. In either case, phantoms cannot occur.

14.4.6 Multiversion locking

The transactions in most database applications are predominantly read-only. Read-only transactions coexist nicely in a database system, because they share locks and never have to wait for each other. However, they do not get along well with update transactions. Suppose that one update transaction is writing to a block. Then all read-only transactions that want to read that block must wait, not just until the block is written, but until the update transaction has completed. Conversely, if an update transaction wants to write a block, it needs to wait until all of the read-only transactions that read the block have completed.

In other words, a lot of waiting is going to occur when read-only and update transactions conflict, regardless of which transaction gets its lock first. Given that this situation is a common one, researchers have developed strategies that can reduce this waiting. One such strategy is called *multiversion locking*.

The principle of multiversion locking

[†] It might need to do this to implement a join query, as we shall see in Chapter 17.

As its name suggests, multiversion locking works by storing multiple versions of each block. The basic idea is as follows.

- Each version of a block is timestamped with the commit time of the transaction that wrote it.
- When a read-only transaction requests a value from a block, the concurrency manager uses the version of the block that was most recently committed at the time the transaction began.

In other words, a read-only transaction sees a snapshot of the committed data as it would have looked at the time the transaction began. Note the term “committed data”. The transaction sees the data that was written by transactions that committed before it began, and does not see the data written by later transactions.

An example of multiversion locking appears in Figure 14-24.

Consider four transactions having the following histories:

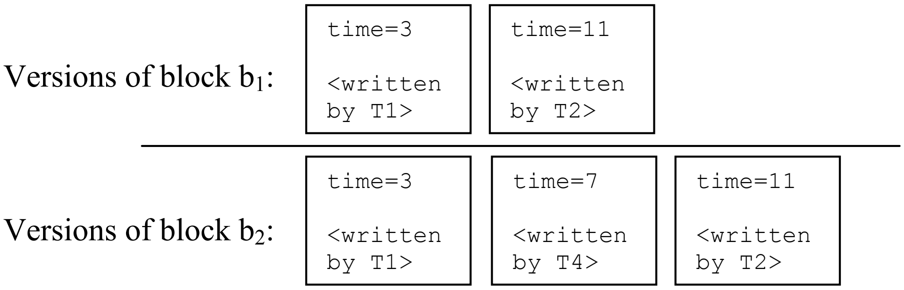
T_1: $W(b_1);\ W(b_2);$
T_2: $W(b_1);\ W(b_2);$
T_3: $R(b_1);\ R(b_2);$
T_4: $W(b_2);$

Suppose that these transactions execute according to the following schedule:

$W_1(b_1);\ W_1(b_2);\ C_1;\ W_2(b_1);\ R_3(b_1);\ W_4(b_2);\ C_4;\ R_3(b_2);\ C_3;\ W_2(b_1);\ C_2;$

In this schedule we assume that a transaction begins where its first operation is, and use C_i to indicate when transaction T_i commits. The update transactions T_1, T_2, and T_4 follow the lock protocol, as you can verify from the schedule. Transaction T_3 is a read-only transaction and does not follow the protocol.

The concurrency manager stores a version of a block for each update transaction that writes to it. Thus there will be two versions of b_1 and three versions of b_2:



The timestamp on each version is the time when its transaction commits, not when the writing occurred. We assume that each operation takes one time unit, so T_1 commits at time 3, T_4 at time 7, T_3 at time 9, and T_2 at time 11.

Now consider the read-only transaction T_3. It begins at time 5, which means that it should see the values that were committed at that point – namely, the changes made by T_1 but not T_2 or T_4. Thus, it will see the versions of b_1 and b_2 that were timestamped at time 3. Note that T_3 will not see the version of b_2 that was timestamped at time 7, even though that version had been committed by the time that the read occurred.

Figure 14-24: An example of multiversion locking

The beauty of multiversion locking is that read-only transactions do not need to obtain locks, and thus *never have to wait*. The concurrency manager chooses the appropriate version of a requested block according to the start time of the transaction. An update transaction can be concurrently making changes to the same block, but a read-only transaction will not care because it sees a different version of that block.

Multi-version locking only applies to read-only transactions. Update transactions need to follow the lock protocol, obtaining both slocks and xlocks as appropriate. The reason is every update transaction reads and writes the current version of the data (and never a previous version), and thus conflicts are possible. But remember that these conflicts are between update transactions only, and not with the read-only transactions. Thus, assuming that there are relatively few update transactions, waiting will be much less frequent.

Implementing multiversion locking

Now that we have seen how multiversion locking should work, let's examine how the concurrency manager does what it needs to do. The basic issue is how to maintain the versions of each block. A straightforward but somewhat difficult approach would be to explicitly save each version in a dedicated "version file". Instead, we shall use a different approach, in which the log can be used to reconstruct any desired version of a block. The implementation works as follows.

The *commit* procedure for an update transaction is revised to include the following actions:

- A timestamp is chosen for this transaction.
- The recovery manager writes the timestamp as part of the transaction's commit log record.
- For each xlock held by the transaction, the concurrency manager pins the block, writes the timestamp to the beginning of the block, and unpins the buffer.

Each read-only transaction is given a timestamp when it starts. Suppose that a transaction having timestamp t requests a block b. The concurrency manager takes the following steps to reconstruct the appropriate version:

- It copies the current version of block b to a new page.
- It reads the log backwards, three times.
    - *It constructs a list of transactions that committed after t.* Since transactions commit in timestamp order, it can stop reading the log when it finds a commit record whose timestamp is less than t.
    - *It constructs a list of uncompleted transactions* by looking for log records written by transactions that do not have a commit or rollback record. It can stop reading the log when it encounters a quiescent checkpoint record or the earliest start record of a transaction in a nonquiescent checkpoint record.
    - *It uses the update records to undo values in the copy of b.* When it encounters an update record for b written by a transaction on either of the above lists, it performs an undo. It can stop reading the log when it encounters the start record for the earliest transaction on the lists.
- The modified copy of b is returned to the transaction.

In other words, the concurrency manager reconstructs the version of the block at time t, by undoing the modifications made by transactions that did not commit before t. This algorithm is somewhat simplified. It is possible to rewrite the algorithm so that the concurrency manager makes a single pass through the log file instead of three passes; see Exercise 14.39.

Finally, we note that a transaction needs to specify whether it is read-only or not, since the concurrency manager treats the two types of transaction differently. In JDBC, this specification is performed by the method *setReadOnly* in the *Connection* interface. For example:

```
Connection conn = ...  // obtain the connection
conn.setReadOnly(true);
```

The call to *setReadOnly* is considered to be a "hint" to the database system. The system can choose to ignore the call if it does not support multiversion locking.

14.4.7 Transaction isolation levels

As we have seen, enforcing serializability causes a considerable amount of waiting, because the lock protocol requires transactions to hold their locks until they complete. Consequently, if a transaction T_1 happens to need just one lock that conflicts with a lock held by T_2, then T_1 cannot do anything else until T_2 completes.

Multiversion locking is very attractive because it allows read-only transactions to execute without locks, and therefore without the inconvenience of having to wait. However, the implementation of multiversion locking is somewhat complex, and requires additional disk accesses to recreate the versions. Moreover, multiversion locking does not apply to transactions that update the database.

There is another way for a transaction to reduce the amount of time it waits for locks – it can specify that it does need complete serializability. In Chapter 8, we saw that JDBC defines four *transaction isolation levels*. Figure 14-25 summarizes these levels and their properties.

ISOLATION LEVEL	PROBLEMS	LOCK USAGE	COMMENTS
serializable	none	slocks held to completion, slock on eof marker	the only level that guarantees correctness
repeatable read	phantoms	slocks held to completion, no slock on eof marker	useful for modify-based transactions
read committed	phantoms, values may change	slocks released early, no slock on eof marker	useful for conceptually separable transactions whose updates are “all or nothing”
read uncommitted	phantoms, values may change, dirty reads	no slocks at all	useful for read-only transactions that tolerate inaccurate results

Figure 14-25: Transaction isolation levels

Chapter 8 related these isolation levels to the various problems that can occur. What is new about Figure 14-25 is that it also relates these levels to the way that slocks are used. Serializable isolation requires very restrictive shared locking, whereas read-uncommitted isolation does not even use slocks. Clearly, the less restrictive the locking, the less waiting that occurs. But less restrictive locking also introduces more inaccuracies into the results of queries: a transaction may see phantoms, or it may see two different values at a location at different times, or it may see values written by an uncommitted transaction.

It is also important to note that these isolation levels apply only to data *reading*. All transactions, regardless of their isolation level, behave correctly with respect to writing data. They must obtain the appropriate xlocks (including the xlock on the eof marker) and hold them to completion. The reason is that an individual transaction may choose to tolerate inaccuracies when it runs a query, but an inaccurate update affects the entire database system and cannot be tolerated.

We now briefly discuss each isolation level.

serializable isolation

This isolation level corresponds to the lock protocol, and is the only level that guarantees the ACID property of *isolation*. In theory, every transaction ought to be run using serializable isolation. In practice, however, serializable transactions tend to be too slow for high-performance applications. Developers who choose non-serializable isolation levels must consider carefully the extent to which inaccurate results will occur, and the acceptability of such inaccuracies.

repeatable read isolation

The first way to reduce the use of slocks is to not slock the end-of-file marker. In this case the transaction will notice when another transaction appends new blocks to the file, and thus may see new, phantom values each time it examines the file.

read committed isolation

A transaction can further reduce the impact of slocks by releasing them early. We examined the problems caused by the early release of slocks back when we discussed the lock protocol. The basic problem is this: Suppose that transaction T_1 releases one slock before obtaining another, and during that interval a transaction T_2 modifies both unlocked blocks. This modification effectively splits T_1 in half – The values it reads before T_2 are consistent, the values it reads after T_2 are consistent, but the values on either half are not consistent with each other.

Although a JDBC transaction can specify read committed isolation, it cannot tell the concurrency manager when to release its slocks. That decision is left to the individual database system. An interesting and useful choice is made by Oracle, which chooses to release the slocks after each SQL statement in the transaction. Consequently, the result of each SQL statement is consistent (modulo phantoms), but the results need not be consistent with each other. In a sense, each statement acts like it would in a separate transaction.

A transaction would choose read committed isolation if it consisted of multiple activities, each of which was conceptually separate from the others. Such a transaction is almost the same as a sequence of smaller transactions; the difference is that the updates performed by the transaction would behave as an atomic, “all or nothing” unit.

read uncommitted isolation

The least restrictive way to use slocks is to not use them. A transaction that uses read uncommitted isolation is therefore exceptionally fast, but also suffers from serious consistency problems. In addition to the problems of the previous isolation levels, the transaction is also able to read modifications made by uncommitted transactions (called “dirty reads”), and therefore has no way of getting an accurate picture of the database.

The read uncommitted isolation level is considered to be inappropriate for update transactions, because of the possibility of updating the database based on data that may not even exist. A read-only transaction would use this isolation level if it were computing information in which approximate answers are acceptable, such as queries that compute statistics. An example of such a query might be “for each year, compute the average of all the grades given in that year". The computation might miss some updated or new grades, but there is a good chance that the results will be reasonably accurate.

How does read uncommitted isolation compare with multiversion locking? Both apply to read-only transactions, and both operate without locks. However, transactions that use read uncommitted isolation see the current value of each block that it reads, regardless of which transaction wrote to it or when. It is not even close to being serializable. On the other hand, a transaction that uses multiversion locking sees the committed contents of the blocks at a single point in time, and is serializable.

14.4.8 Data item granularity

We have discussed concurrency management in terms of locking blocks. But like recovery, other locking granularities are possible: The concurrency manager could lock values, files, or even the entire database. We call a unit of locking a *concurrency data item*.

> *A concurrency data item is the unit of locking*
> *used by the concurrency manager.*

The principles of concurrency control are not affected by the granularity of data item used. All of our definitions, protocols, and algorithms apply to any data item. The choice of granularity is therefore a practical one, which needs to balance efficiency with flexibility. This section examines some of these tradeoffs.

The concurrency manager keeps a lock for each data item. A smaller granularity size is useful because it allows for more concurrency. For example, suppose that two transactions wish to concurrently modify different parts of the same block. These concurrent modifications are possible with value-granularity locking, but not with block-granularity locking.

However, a smaller granularity requires more locks. Values tend to make impractically small data items, because they entail an enormous number of locks. At the other extreme, using files as data items would require very few locks, but would also significantly impact concurrency – A client would need to xlock the entire file in order to update any portion of it. Using blocks as data items is a reasonable compromise.

As an aside, we note that some operating systems (such as MacOS and Windows) use file-granularity locking to implement a primitive form of concurrency control. In particular, an application cannot write to a file without an xlock on the file, and it cannot obtain the xlock if that file is currently being used by another application.

Some concurrency managers support data items at multiple granularities, such as both blocks and files. A transaction that plans to access only a few blocks of a file could lock them separately; but if the transaction plans to access all (or most of) the file, it would obtain a single file-granularity lock. This approach blends the flexibility of small-granularity items with the convenience of high-level items.

Another possible granularity is to use *data records* as concurrency data items. Data records are handled by the record manager, which is the topic of the next chapter. In this book we have structured the database system so that the concurrency manager is below the record manager; thus, the concurrency manager does not understand records and cannot lock them. However, some commercial systems (such as Oracle) are built so that the concurrency manager is above the record manager and calls its methods. In this case, data records would be the most natural concurrency data item.

Although data-record granularity appears attractive, it introduces additional problems with phantoms. Since new data records can get inserted into existing blocks, a transaction that reads all records from a block needs a way to keep other transactions from inserting records into that block. The solution is for the concurrency manager to also support a coarser-granularity data item, such as blocks or files. In fact, some commercial systems avoid phantoms by simply forcing a transaction to obtain an xlock on the file before it performs any insertion.

14.4.9 The SimpleDB concurrency manager

The SimpleDB concurrency manager is implemented via the class *ConcurrencyMgr* in the package *simpledb.tx.concurrency*. The concurrency manager implements the lock protocol, using block-level granularity. Its API appears in Figure 14-26.

ConcurrencyMgr

```
public ConcurrencyMgr(int txnum);
public void sLock(Block blk);
public void xLock(Block blk);
public void release();
```

Figure 14-26: The API for the SimpleDB concurrency manager

Each transaction creates its own concurrency manager. The methods of the concurrency manager are similar to those of the lock table, but are transaction-specific. Each *ConcurrencyMgr* object keeps track of the locks held by its transaction. The methods *sLock* and *xLock* request a lock from the lock table only if the transaction does not yet have it. The method *release* is called at the end of the transaction to unlock all of its locks.

The *ConcurrencyMgr* class makes use of the class *LockTable*, which is the SimpleDB lock table. The remainder of this section examines the implementation of these two classes.

The class *LockTable*

The code for the class *LockTable* appears in Figure 14-27.

```
class LockTable {
   private static final long MAX_TIME = 10000; // 10 seconds

   private Map<Block,Integer> locks = new HashMap<Block,Integer>();

   public synchronized void sLock(Block blk) {
      try {
         long timestamp = System.currentTimeMillis();
         while (hasXlock(blk) && !waitingTooLong(timestamp))
            wait(MAX_TIME);
         if (hasXlock(blk))
            throw new LockAbortException();
         int val = getLockVal(blk);  // will not be negative
         locks.put(blk, val+1);
      }
      catch(InterruptedException e) {
         throw new LockAbortException();
      }
   }

   public synchronized void xLock(Block blk) {
      try {
         long timestamp = System.currentTimeMillis();
         while (hasOtherSLocks(blk) && !waitingTooLong(timestamp))
            wait(MAX_TIME);
         if (hasOtherSLocks(blk))
            throw new LockAbortException();
         locks.put(blk, -1);
      }
      catch(InterruptedException e) {
         throw new LockAbortException();
      }
   }

   public synchronized void unlock(Block blk) {
      int val = getLockVal(blk);
      if (val > 1)
         locks.put(blk, val-1);
      else {
         locks.remove(blk);
         notifyAll();
      }
   }

   private boolean hasXlock(Block blk) {
      return getLockVal(blk) < 0;
   }

   private boolean hasOtherSLocks(Block blk) {
      return getLockVal(blk) > 1;
   }

   private boolean waitingTooLong(long starttime) {
      return System.currentTimeMillis() - starttime > MAX_TIME;
   }
   private int getLockVal(Block blk) {
      Integer ival = locks.get(blk);
      return (ival == null) ? 0 : ival.intValue();
   }
}
```

Figure 14-27: The code for the SimpleDB class *LockTable*

The *LockTable* object holds a *Map* variable called *locks*. This map contains an entry for each block that currently has an assigned lock. The value of an entry will be an *Integer* object; a value of -1 denotes that an exclusive lock is assigned, whereas a positive value denotes the current number of shared locks assigned.

The *sLock* and *xLock* methods work very similarly to the *pin* and *pinNew* methods of *BufferMgr*. Each method calls the Java *wait* method inside of a loop, which means that the client thread is continually placed on the wait list as long as the loop condition holds. The loop condition for *sLock* calls the method *hasXlock*, which returns *true* iff the block has an entry in *locks* with a value of -1. The loop condition for *xLock* calls the method *hasOtherLocks*, which returns *true* if the block has an entry in *locks* with a value greater than 1. The rationale is that the concurrency manager will always obtain an slock on the block before requesting the xlock, and so a value higher than 1 indicates that some other transaction also has a lock on this block.

The *unlock* method either removes the specified lock from the *locks* collection (if it is either an exclusive lock or a shared lock held by only one transaction), or decrements the number of transactions still sharing the lock. If the lock is removed from the collection, the method calls the Java *notifyAll* method, which moves all waiting threads to the ready list for scheduling. The internal Java thread scheduler resumes each thread in some unspecified order. There may be several threads waiting on the same released lock. By the time a thread is resumed, it may discover that the lock it wants is unavailable, and will place itself on the wait list again.

This code is not especially efficient about how it manages thread notification. The *notifyAll* method moves *all* waiting threads, which includes threads waiting on other locks. Those threads, when scheduled, will (of course) discover that their lock is unavailable, and will place themselves back on the wait list. On one hand, this strategy will not be too costly if there are relatively few conflicting database threads running concurrently. On the other hand, a database system ought to be more sophisticated than that. Exercises 14.50-14.51 ask you to improve the wait/notification mechanism.

The class *ConcurrencyMgr*

The code for the class *ConcurrencyMgr* appears in Figure 14-28.

```
public class ConcurrencyMgr {

   private static LockTable locktbl = new LockTable();
   private Map<Block,String> locks  = new HashMap<Block,String>();

   public void sLock(Block blk) {
      if (locks.get(blk) == null) {
         locktbl.sLock(blk);
         locks.put(blk, "S");
      }
   }

   public void xLock(Block blk) {
      if (!hasXLock(blk)) {
         sLock(blk);
         locktbl.xLock(blk);
         locks.put(blk, "X");
      }
   }

   public void release() {
      for (Block blk : locks.keySet())
         locktbl.unlock(blk);
      locks.clear();
   }

   private boolean hasXLock(Block blk) {
      String locktype = locks.get(blk);
      return locktype != null && locktype.equals("X");
   }
}
```

Figure 14-28: The code for the SimpleDB class *ConcurrencyMgr*

Although there is a concurrency manager for each transaction, they all need to use the same lock table. This requirement is implemented by having each *ConcurrencyMgr* object share a static *LockTable* variable. The description of the locks held by the transaction is held in the local variable *locks*. This variable holds a map that has an entry for each locked block. The value associated with the entry is either “S” or “X”, depending on whether there is an slock or an xlock on that block.

The method *sLock* first checks to see if the transaction already has a lock on the block; if so, there is no need to go to the lock table. Otherwise, it calls the lock table’s *sLock* method, and waits for the lock to be granted. The method *xLock* need not do anything if the transaction already has an xlock on the block. If not, the method first obtains an slock on the block, and then obtains the xlock. (Recall that the lock table’s *xLock* method assumes that the transaction already has an slock.)

14.5 Implementing SimpleDB Transactions

Section 14.2 introduced the API for the class *Transaction*. We have finally come to the point where we can discuss its implementation.

The *Transaction* class makes use of the class *BufferList*, which manages the buffers it has pinned. We shall discuss each class in turn.

<u>The class *Transaction*</u>

The code for class *Transaction* appears in Figure 14-29. Each *Transaction* object creates its own recovery manger and concurrency manager. It also creates the object *myBuffers* to manage the currently-pinned buffers.

```
public class Transaction {
   private static int nextTxNum = 0;
   private RecoveryMgr    recoveryMgr;
   private ConcurrencyMgr concurMgr;
   private int txnum;
   private BufferList myBuffers = new BufferList();

   public Transaction() {
      txnum       = nextTxNumber();
      recoveryMgr = new RecoveryMgr(txnum);
      concurMgr   = new ConcurrencyMgr();
   }

   public void commit() {
      myBuffers.unpinAll(txnum);
      recoveryMgr.commit();
      concurMgr.release();
      System.out.println("transaction " + txnum + " committed");
   }

   public void rollback() {
      myBuffers.unpinAll(txnum);
      recoveryMgr.rollback();
      concurMgr.release();
      System.out.println("transaction " + txnum + " rolled back");
   }

   public void recover() {
      recoveryMgr.recover();
   }

   public void pin(Block blk) {
      myBuffers.pin(blk);
   }

   public void unpin(Block blk) {
      myBuffers.unpin(blk);
   }

   public int getInt(Block blk, int offset) {
      concurMgr.sLock(blk);
      Buffer buff = myBuffers.getBuffer(blk);
      return buff.getInt(offset);
   }

   public String getString(Block blk, int offset) {
      concurMgr.sLock(blk);
      Buffer buff = myBuffers.getBuffer(blk);
      return buff.getString(offset);
   }

   public void setInt(Block blk, int offset, int val) {
      concurMgr.xLock(blk);
      Buffer buff = myBuffers.getBuffer(blk);
      int lsn = recoveryMgr.setInt(buff, offset, val);
      buff.setInt(offset, val, txnum, lsn);
   }

   public void setString(Block blk, int offset, String val) {
      concurMgr.xLock(blk);
      Buffer buff = myBuffers.getBuffer(blk);
      int lsn = recoveryMgr.setString(buff, offset, val);
      buff.setString(offset, val, txnum, lsn);
   }

   public int size(String filename) {
      Block dummyblk = new Block(filename, -1);
      concurMgr.sLock(dummyblk);
      return SimpleDB.fileMgr().size(filename);
   }

   public Block append(String filename, PageFormatter fmtr) {
      Block dummyblk = new Block(filename, -1);
      concurMgr.xLock(dummyblk);
      Block blk = myBuffers.pinNew(filename, fmtr);
      unpin(blk);
      return blk;
   }

   private static synchronized int nextTxNumber() {
      nextTxNum++;
      System.out.println("new transaction: " + nextTxNum);
      return nextTxNum;
   }
}
```

Figure 14-29: The code for the SimpleDB class *Transaction*

The *commit* and *rollback* methods perform the following activities:

- they unpin any remaining buffers;
- they call the recovery manager to commit (or roll back) the transaction;
- they call the concurrency manager to release its locks.

The methods *getInt* and *getString* first acquire an slock on the specified block from the concurrency manager, and then return the requested value from the buffer. The methods *setInt* and *setString* first acquire an xlock from the concurrency manager; they then call the corresponding method in the recovery manager to create the appropriate log record and return its LSN. This LSN can then be passed to the buffer's *setInt* (or *setString*) method.

The methods *size* and *append* treat the end-of-file marker as a "dummy" block with block number -1. The method *size* obtains an slock on the block, and *append* obtains an xlock on the block.

The class *BufferList*

The class *BufferList* manages the list of currently-pinned buffers for a transaction; see Figure 14-30.

```
class BufferList {
   private Map<Block,Buffer> buffers = new HashMap<Block,Buffer>();
   private List<Block> pins = new ArrayList<Block>();
   private BufferMgr bufferMgr = SimpleDB.bufferMgr();

   Buffer getBuffer(Block blk) {
      return buffers.get(blk);
   }

   void pin(Block blk) {
      Buffer buff = bufferMgr.pin(blk);
      buffers.put(blk, buff);
      pins.add(blk);
   }

   Block pinNew(String filename, PageFormatter fmtr) {
      Buffer buff = bufferMgr.pinNew(filename, fmtr);
      Block blk = buff.block();
      buffers.put(blk, buff);
      pins.add(blk);
      return blk;
   }

   void unpin(Block blk) {
      Buffer buff = buffers.get(blk);
      bufferMgr.unpin(buff);
      pins.remove(blk);
      if (!pins.contains(blk))
         buffers.remove(blk);
   }

   void unpinAll(int txnum) {
      for (Block blk : pins) {
         Buffer buff = buffers.get(blk);
         bufferMgr.unpin(buff);
      }
      buffers.clear();
      pins.clear();
   }
}
```

Figure 14-30: The code for the SimpleDB class *BufferList*

A *BufferList* object needs to know two things: which buffer is assigned to a specified block; and how many times each block is pinned. The code uses a map to determine buffers, and a list to determine pin counts. The list contains a *Block* object as many times as it is pinned; each time the block is unpinned, one copy of it is removed from the list.

The method *unpinAll* performs the buffer-related activity required when a transaction commits or rolls back – it has the buffer manager flush all buffers modified by the transaction, and unpins any still-pinned buffers.

14.6 Testing Transactions

In order to unit test transaction-based code, it is necessary to run several transactions concurrently. The Java *Thread* class makes it easy to run multiple threads. For example, the demo code *TransactionTest.java* in Figure 14-31 creates three threads that implement the locking scenario of Figure 14-18. The output of this program shows the order in which the locks are both attempted and granted. Test drivers for other parts of the recovery and concurrency managers are left for Exercise 14.58.

```
public class TransactionTest {
   public static void main(String[] args) {
      //initialize the database system
      SimpleDB.init(args[0]);

      TestA t1 = new TestA(); new Thread(t1).start();
      TestB t2 = new TestB(); new Thread(t2).start();
      TestC t3 = new TestC(); new Thread(t3).start();
   }
}

class TestA implements Runnable {
   public void run() {
      try {
         Transaction tx = new Transaction();
         Block blk1 = new Block("junk", 1);
         Block blk2 = new Block("junk", 2);
         tx.pin(blk1);
         tx.pin(blk2);
         System.out.println("Tx A: read 1 start");
         tx.getInt(blk1, 0);
         System.out.println("Tx A: read 1 end");
         Thread.sleep(1000);
         System.out.println("Tx A: read 2 start");
         tx.getInt(blk2, 0);
         System.out.println("Tx A: read 2 end");
         tx.commit();
      }
      catch(InterruptedException e) {};
   }
}

class TestB implements Runnable {
   public void run() {
      try {
         Transaction tx = new Transaction();
         Block blk1 = new Block("junk", 1);
         Block blk2 = new Block("junk", 2);
         tx.pin(blk1);
         tx.pin(blk2);
         System.out.println("Tx B: write 2 start");
         tx.setInt(blk2, 0, 0);
         System.out.println("Tx B: write 2 end");
         Thread.sleep(1000);
         System.out.println("Tx B: read 1 start");
         tx.getInt(blk1, 0);
         System.out.println("Tx B: read 1 end");
         tx.commit();
      }
      catch(InterruptedException e) {};
   }
}

class TestC implements Runnable {
   public void run() {
      try {
         Transaction tx = new Transaction();
         Block blk1 = new Block("junk", 1);
         Block blk2 = new Block("junk", 2);
         tx.pin(blk1);
         tx.pin(blk2);
         System.out.println("Tx C: write 1 start");
         tx.setInt(blk1, 0, 0);
         System.out.println("Tx C: write 1 end");
         Thread.sleep(1000);
         System.out.println("Tx C: read 2 start");
         tx.getInt(blk2, 0);
         System.out.println("Tx C: read 2 end");
         tx.commit();
      }
      catch(InterruptedException e) {};
   }
}
```

Figure 14-31: Test code for the lock table

14.7 Chapter Summary

- Data can get lost or corrupted when client programs are able to run indiscriminately. Database systems force client programs to consist of *transactions*.

- A transaction is a group of operations that behaves as a single operation. It satisfies the ACID properties of *atomicity*, *consistency*, *isolation* and *durability*.

- The *recovery manager* is responsible for ensuring atomicity and durability. It is the portion of the server that reads and processes the log. It has three functions: to write log records, to roll back a transaction, and to recover the database after a system crash.

- Each transaction writes a *start record* to the log to denote when it begins, *update records* to indicate the modifications it makes, and a *commit* or *rollback record* to denote when it completes. In addition, the recovery manager can write *checkpoint records* to the log at various times.

- The recovery manager *rolls back* a transaction by reading the log backwards. It uses the transaction's update records to undo the modifications.

- The recovery manager *recovers* the database after a system crash.

- The *undo-redo recovery algorithm* undoes the modifications made by uncommitted transactions, and redoes the modifications made by committed transactions.

- The *undo-only recovery algorithm* assumes that modifications made by a committed transaction are flushed to the disk before the transaction commits. Thus it only needs to undo modifications made by uncommitted transactions.

- The *redo-only recovery algorithm* assumes that modified buffers are not flushed until the transaction commits. This algorithm requires a transaction to keep modified buffers pinned until it completes, but it avoids the need to undo uncommitted transactions.

- The *write-ahead logging* strategy requires that an update log record be forced to disk before the modified data page. Write-ahead logging guarantees that modifications to the database will always be in the log and therefore will always be undoable.

- Checkpoint records are added to the log in order to reduce the portion of the log that the recovery algorithm needs to consider. A *quiescent checkpoint record* can be written when no transactions are currently running; a *nonquiescent checkpoint record* can be written at any time. If undo-redo (or redo-only) recovery is used, then the recovery manager must flush modified buffers to disk before it writes a checkpoint record.

- A recovery manager can choose to log values, records, pages, files, etc. The unit of logging is called a *recovery data item*. The choice of data item involves a tradeoff: A large-granularity data item will require fewer update log records, but each log record will be larger.

- The *concurrency manager* is the portion of the server that is responsible for the correct execution of concurrent transactions.

- The sequence of operations performed by the transactions in a system is called a *schedule*. A schedule is *serializable* if it is equivalent to a serial schedule. Only serializable schedules are correct.

- The concurrency manager uses locking to guarantee that schedules are serializable. In particular, it requires all transactions to follow the *lock protocol*, which states:
  - Before reading a block, acquire a shared lock on it.
  - Before modifying a block, acquire an exclusive lock on it.
  - Release all locks after commit or rollback.

- A *deadlock* can occur if there is a cycle of transactions where each transaction is waiting for a lock held by the next transaction. The concurrency manager can detect deadlock by keeping a *waits-for* graph and checking for cycles.

- The concurrency manager can also use algorithms to approximate deadlock detection. The *wait-die algorithm* forces a transaction to rollback if it needs a lock held by an older transaction. The *time-limit algorithm* forces a transaction to rollback if it has been waiting for a lock longer than expected. Both of these algorithms will remove deadlock when it exists, but might also rollback a transaction unnecessarily.

- While one transaction is examining a file, another transaction might append new blocks to it. The values in those blocks are called *phantoms*. Phantoms are undesirable because they violate serializability. A transaction can avoid phantoms by locking the end-of-file marker.

- The locking needed to enforce serializability significantly reduces concurrency. The *multiversion locking* strategy allows read-only transactions to run without locks (and thus without having to wait). The concurrency manager implements multiversion locking by associating timestamps with each transaction, and using those timestamps to reconstruct the version of the blocks as they were at a specified point in time.

- Another way to reduce the waiting time imposed by locking is to remove the requirement of serializability. A transaction can specify that it belongs to one of four *isolation levels*: *serializable*, *repeatable read*, *read committed*, or *read uncommitted*. Each non-serializable isolation level reduces the restrictions on slocks given by the log protocol, and results in less waiting as well as increased severity of read problems. Developers who choose non-serializable isolation levels must consider carefully the extent to which inaccurate results will occur, and the acceptability of such inaccuracies.

- As with recovery, a concurrency manager can choose to lock values, records, pages, files, etc. The unit of locking is called a *concurrency data item*. The choice of data item involves a tradeoff. A large-granularity data item will require fewer locks, but the larger locks will conflict more readily and thus reduce concurrency.

14.8 Suggested Reading

The notion of a transaction is fundamental to many areas of distributed computing, not just database systems. Researchers have developed an extensive set of techniques and algorithms; the ideas in this chapter are only the small tip of a very large iceberg. Two excellent books that provide an overview of the field are [Bernstein and Newcomer 1997] and [Gray and Reuter 1993]. A comprehensive treatment of many concurrency control and recovery algorithms appears in [Bernstein et al. 1987]. A widely-adopted recovery algorithm is called ARIES, and is described in [Mohan et al. 1992].

Oracle's implementation of the serializable isolation level is called *Snapshot Isolation*. Its advantage is that it is more efficient than our locking protocol. However, it does not guarantee serializability. Although most schedules will be serializable, there are certain scenarios in which is can result in non-serializable behavior. The article [Fekete et al. 2005] analyzes these scenarios and shows how to modify at-risk applications to guarantee serializability.

It is often useful to think of a transaction as being comprised of several smaller, coordinated transactions. For example in a *nested transaction*, a parent transaction is able to spawn one or more child subtransactions; when a subtransaction completes, its parent decides what to do. If the subtransaction aborts, the parent could choose to abort all of its children, or it might continue by spawning another transaction to replace the aborted one. The basics of nested transactions can be found in [Moss 1985]. The article [Weikum 1991] defines *multi-level transactions*, which are similar to nested transactions; the difference is that a multi-level transaction uses subtransactions as a way to increase efficiency via parallel execution.

14.9 Exercises

CONCEPTUAL EXERCISES

14.1 Consider Figure 14-1, and assume it is being run without transactions. Give a scenario in which two seats are reserved for a customer but only one sale is recorded.

14.2 Software configuration managers such as CVS or Subversion allow a user to commit a series of changes to a file and to rollback a file to a previous state. They also allow multiple users to modify a file concurrently.
a) What is the notion of a transaction in such systems?
b) How do such systems ensure serializability?
c) Would such an approach work for a database system? Explain.

14.3 Consider a JDBC program that performs several unrelated SQL queries but does not modify the database. The programmer decides that since nothing is updated, the concept of a transaction is unimportant; thus the entire program is run as a single transaction.
a) Explain why the concept of a transaction *is* important to a read-only program.
b) What is the problem with running the entire program as a large transaction?
c) How much overhead is involved in committing a read-only transaction? Does it make sense for the program to commit after every SQL query?

14.4 The recovery manager writes a start record to the log when each transaction begins.
a) What is the practical benefit of having start records in the log?

b) Suppose that a database system decides not to write start records to the log. Can the recovery manager still function properly? What capabilities are impacted?

14.5 The SimpleDB *rollback* method writes the rollback log record to disk before it returns. Is this necessary? Is it a good idea?

14.6 Suppose that the recovery manager was modified so that it didn't write rollback log records when it finished. Would there be a problem? Would this be a good idea?

14.7 Consider the undo-only commit algorithm of Figure 14-7. Explain why it would be incorrect to swap steps 1 and 2 of the algorithm.

14.8 Show that if the system crashes during a rollback or a recovery, then redoing the rollback (or recovery) is still correct.

14.9 Is there any reason to log the changes made to the database during rollback or recovery? Explain.

14.10 A variation on the nonquiescent checkpointing algorithm is to mention only one transaction in the checkpoint log record, namely the oldest active transaction at the time.
a) Explain how the recovery algorithm will work.
b) Compare this strategy with the strategy given in the text. Which is simpler to implement? Which is more efficient?

14.11 What should the rollback method do if it encounters a quiescent checkpoint log record? What if it encounters a nonquiescent log record? Explain.

14.12 The algorithm for non-quiescent checkpointing does not allow new transactions to start while it is writing the checkpoint record. Explain why this restriction is important for correctness.

14.13 Some algorithms write two records to the log when doing a nonquiescent checkpoint. The first record is `<BEGIN_CKPT>`, and contains nothing else. The second record is the standard <CKPT ...> record, which contains the list of active transactions. The first record is written as soon as the recovery manager decides to do a checkpoint. The second record is written later, after the list of active transactions has been created.
a) Explain why this strategy solves the problem of Exercise 14.12.
b) Give a revised recovery algorithm that incorporates this strategy.

14.14 Explain why the recovery manager will never encounter more than one quiescent checkpoint record during recovery.

14.15 Give an example showing that the recovery manager could encounter several nonquiescent checkpoint records during recovery. What is the best way for it to handle the nonquiescent checkpoint records it finds after the first one?

14.16 Explain why the recovery manager could not encounter a nonquiescent and a quiescent checkpoint record during recovery.

14.17 Consider the recovery algorithm of Figure 14-6. Step 1c doesn't undo a value for transactions that have been rolled back.
a) Explain why this is the correct thing to do.
b) Would the algorithm be correct if it did undo those values? Explain.

14.18 When the *rollback* method needs to restore the original contents of a value, it writes the page directly, and doesn't request any kind of lock. Can this cause a nonserializable conflict with another transaction? Explain.

14.19. Explain why it is not possible to have a recovery algorithm that combines the techniques of undo-only and redo-only recovery. That is, explain why it is necessary to keep either undo information or redo information.

14.20 Suppose that the recovery manager finds the following records in the log file when the system restarts after a crash.

```
<START, 1>
<START, 2>
<SETSTRING, 2, junk, 33, 0, abc, def>
<SETSTRING, 1, junk, 44, 0, abc, xyz>
<START, 3>
<COMMIT, 2>
<SETSTRING, 3, junk, 33, 0, def, joe>
<START, 4>
<SETSTRING, 4, junk, 55, 0, abc, sue>
<NQCKPT, 1, 3, 4>
<SETSTRING, 4, junk, 55, 0, sue, max>
<START, 5>
<COMMIT, 4>
```

a) Assuming undo-redo recovery, indicate what changes to the database will be performed.
b) Assuming undo-only recovery, indicate what changes to the database will be performed.
c) Is it possible for transaction T_1 to have committed, even though it has no commit record in the log?
d) Is it possible for transaction T_1 to have modified a buffer containing block 23?
e) Is it possible for transaction T_1 to have modified block 23 on disk?
f) Is it possible for transaction T_1 to have not modified a buffer containing block 44?

14.21 Is a serial schedule always serializable? Is a serializable schedule always serial? Explain.

14.22 This exercise asks you to examine the need for non-serial schedules.

a) Suppose that the database is much larger than the size of the buffer pool. Explain why the database system will handle transactions more quickly if it can execute the transactions concurrently.
b) Conversely, explain why concurrency is less important if the database fits into the buffer pool.

14.23 The *get/set* methods in the SimpleDB class *Transaction* obtain a lock on the specified block. Why don't they unlock the block when they are done?

14.24 Consider Figure 14-3. Give the history of the transaction if files are the element of concurrency.

14.25 Consider the following two transactions and their histories:

T_1: W(b_1); R(b_2); W(b_1); R(b_3); W(b_3); R(b_4); W(b_2);
T_2: R(b_2); R(b_3); R(b_1); W(b_3); R(b_4); W(b_4);

a) Give a serializable non-serial schedule for these transactions.
b) Add lock and unlock actions to these histories that satisfy the lock protocol.
c) Give a non-serial schedule corresponding to these locks that deadlocks.
d) Show that there is no non-deadlocked non-serial serializable schedule for these transactions that obeys the lock protocol.

14.26 Give an example schedule which is serializable, but has conflicting write-write operations that do not affect the order in which the transactions commit. (HINT: Some of the conflicting operations will not have corresponding read operations.)

14.27 Show that if all transactions obey the 2-phase locking protocol, then all schedules are serializable.

14.28 Show that the waits-for graph has a cycle if and only if there is a deadlock.

14.29 Suppose that a transaction manager maintains a waits-for graph in order to accurately detect deadlocks. In Section 14.4.4 we suggested that the transaction manager roll back the transaction whose request caused the cycle in the graph. Other possibilities are to roll back the oldest transaction in the cycle, the newest transaction in the cycle, the transaction holding the most locks, or the transaction holding the fewest locks. Which possibility makes the most sense to you? Explain.

14.30 Suppose in SimpleDB that transaction T currently has a shared lock on a block, and calls *setInt* on it. Give a scenario in which this will cause a deadlock.

14.31 Consider the example of Section 14.4.2 involving threads A, B, and C. Give a schedule that causes deadlock.

14.32 Consider the locking scenario (involving threads A, B, and C) that is described in Section 14.4.2. Draw the different states of the waits-for graph as locks are requested and released.

14.33 A variant of the *wait-die* protocol is called *wound-wait*, and is as follows:

- If T_1 has a lower number than T_2, then T_2 is aborted (i.e. T_1 “wounds” T_2).
- If T_1 has a higher number than T_2, then T_1 waits for the lock.

The idea is that if an older transaction needs a lock held by a younger one, then it simply kills the younger one and takes the lock.
a) Show that this protocol prevents deadlock.
b) Compare the relative benefits of the *wait-die* and *wound-wait* protocols.

14.34 In the *wait-die* deadlock detection protocol, a transaction is aborted if it requests a lock held by an older transaction. Suppose we modified the protocol so that transactions are aborted instead if they request a lock held by a younger transaction. This protocol would also detect deadlocks. How does this revised protocol compare to the original one? Which would you prefer the transaction manager to use? Explain.

14.35 Explain why the lock/unlock methods in class *LockTable* are synchronized. What bad thing could happen if they were not?

14.36 Suppose that a database system uses files as concurrency elements. Explain why phantoms are not possible.

14.37 Give an algorithm for deadlock detection that also handles transactions waiting for buffers.

14.38 Rewrite the algorithm for multiversion locking so that the concurrency manager only makes one pass through the log file.

14.39 The read-committed transaction isolation level purports to reduce a transaction’s waiting time by releasing its slocks early. At first glance, it is not obvious why a transaction would wait less by releasing locks that it already has. Explain the advantages of early lock release, and give illustrative scenarios.

14.40 The method *nextTransactionNumber* is the only method in *Transaction* that is synchronized. Explain why synchronization is not necessary for the other methods.

14.41 Consider the SimpleDB class *Transaction*.
a) Can a transaction pin a block without locking it?
b) Can a transaction lock a block without pinning it?

PROGRAMMING EXERCISES

14.42 In SimpleDB, a transaction acquires an slock on a block whenever a *getInt* or *getString* method is called. Another possibility is for the transaction to acquire the slock when the block is pinned, under the assumption that you don't pin a block unless you intend to look at its contents.
a) Implement this strategy.
b) Compare the benefits of this strategy with that of SimpleDB. Which do you prefer and why?

14.43 After recovery, the log is not needed except for archival purposes. Revise the SimpleDB code so that the log file is saved to a separate directory after recovery, and a new empty log file is begun.

14.44 Revise the SimpleDB recovery manager so that it undoes an update record only when necessary.

14.45 Revise SimpleDB so that it uses blocks as the elements of recovery. A possible strategy is to save a copy of a block the first time a transaction modifies it. The copy could be saved in a separate file, and the update log record could hold the block number of the copy. You will also need to write methods that can copy blocks between files.

14.46 Implement a static method in class *Transaction* that performs quiescent checkpointing. Decide how the method will get invoked (e.g. every n transactions, every n seconds, or manually). You will need to revise *Transaction* as follows:

- Use a static variable to hold all currently active transactions.
- Revise the constructor of the *Transaction* to check to see if a checkpoint is being performed, and if so to place itself on a wait list until the checkpoint procedure completes.

14.47 Implement non-quiescent checkpointing, using the strategy described in the text.

14.48 Suppose a transaction appends a lot of blocks to a file, writes a bunch of values to these blocks, and then rolls back. The new blocks will be restored to their initial condition, but they themselves will not be deleted from the file. Modify SimpleDB so that they will. (HINT: You can take advantage of the fact that only one transaction at a time can be appending to a file, which means that the file can be truncated during rollback. You will need to add to the file manager the ability to truncate a file.)

14.49 Log records could also be used for auditing a system as well as recovery. For auditing, the record needs to store the date when the activity occurred, as well as the IP address of the client.
a) Revise the SimpleDB log records in this way.
b) Design and implement a class whose methods support common auditing tasks, such as finding when a block was last modified, or what activity occurred by a particular transaction or from a particular IP address.

14.50 The recovery algorithm used by SimpleDB undoes values of rolled-back transactions. In Exercise 14.17, we saw that this was unnecessary. Revise SimpleDB so that it does not undo values of rolled-back transactions.

14.51 Revise SimpleDB so that it uses undo-redo recovery.

14.52 Implement deadlock detection in SimpleDB using:
a) the *wait-die* protocol given in the text;
b) the *wound-wait* protocol given in Exercise 14.33.

14.53 Revise the lock table so that is uses individual wait lists for each block. (So *notifyAll* only touches the threads waiting on the same lock.)

14.54 Revise the lock table so that it keeps its own explicit wait list(s), and chooses itself which transactions to notify when a lock becomes available. (That is, it uses the Java method *notify* instead of *notifyAll*.)

14.55 Revise the SimpleDB concurrency manager so that:
a) Files are the elements of concurrency.
b) Values are the elements of concurrency. (WARNING: You will still need to keep the methods *size* and *append* from causing conflicts.)

14.56 Each time the server starts up, transaction numbers begin again at 0. This means that throughout the history of the database, there will be multiple transactions having the same number.
a) Explain why this non-uniqueness of transaction numbers is not a significant problem.
b) Revise SimpleDB so that transaction numbers continue from the last time the server was running.

14.57 Write a test driver:
a) to verify that the log manager works (commit, rollback, and recovery).
b) to more completely test the lock manager.
c) to test the entire transaction manager.

15

RECORD MANAGEMENT

and the package simpledb.record

The transaction manager can read a value from a specified location on a disk block, and can write a value to a location on a block. Given the blocks of a file, however, it has no idea what values are in those blocks, nor where those values might be located. The portion of the database system that is responsible for giving structure to a file is called the *record manager*. It organizes a file into a collection of records, and has methods for iterating through the records and placing values in them. In this chapter we study the functionality provided by the record manager, and the techniques used to implement that functionality.

15.1 The Record Manager

> *The record manager is the portion of the database system that is responsible for storing records in a file.*

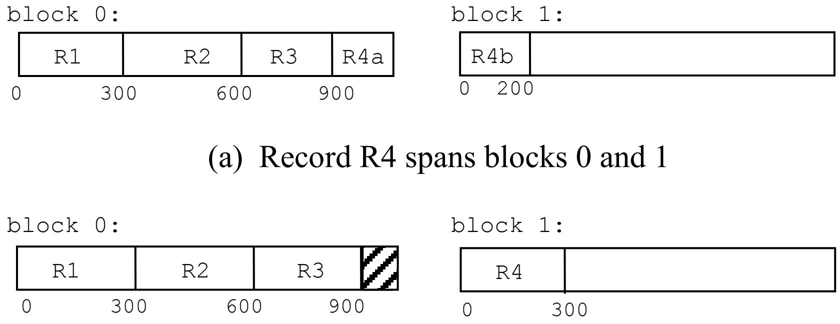
When we design a record manager, we need to addres the following issues:

- Should each record be placed entirely within one block?
- Will all of the records in a block be from the same table?
- Is each field representable using a fixed number of bytes?
- Where should each field value in a record be stored?

This section discusses these issues.

15.1.1 Spanned vs. unspanned records

Suppose that the record manager needs to insert four 300-byte records into a file, where the block size is 1000 bytes. Three records fit nicely into the first 900 bytes of the block. But what should the record manager do with the fourth record? There are two options, as shown in Figure 15-1.



(a) Record R4 spans blocks 0 and 1

(b) Record R4 is stored entirely in block 1

Figure 15-1: Spanned vs. unspanned records

In Figure 15-1(a), the record manager creates a *spanned record* – that is, it stores the first 100 bytes of the record in the existing block, and the next 200 bytes of the record in a new block. In Figure 15-1(b), the record manager stores the entire fourth record in a new block.

A spanned record is a record whose values span two or more blocks.

The record manager has to decide whether to create spanned records or not. One advantage to creating spanned records is that no disk space is wasted. In Figure 15-1(b), 100 bytes (or 10%) of each block is wasted. An even worse case would be if each record contained 501 bytes – then a block could contain only 1 record, and nearly 50% of its space would be wasted.

Another advantage to using spanned records is that the size of the record is not bound by the block size. If records can be larger than a block, then spanning is necessary.

The disadvantage to creating spanned records is that they increase the complexity of the record manager. An unspanned record is simple to deal with, because its data lies completely within a single block. A spanned record, however, will be split up among several blocks, which means that multiple blocks will need to be read and the record will need to be reconstructed, perhaps by reading it into a separate area.

15.1.2 Homogeneous vs. non-homogeneous files

A file is homogeneous if all of its records come from the same table.

The record manager must decide whether or not to allow non-homogeneous files. The tradeoff is again one of efficiency vs. flexibility.

For example, consider the STUDENT and DEPT tables from Figure 1-1. A homogeneous implementation would place all STUDENT records in one file and all DEPT records in another file. This placement makes single-table SQL queries easy to answer – the record manager needs to scan only through the blocks of a single file. However, multi-table queries are less efficient. Consider a query that joins these two tables, such as "Find the names of students and their major departments". The record manager will have to search back and forth between the blocks of STUDENT records and the blocks of DEPT records (as we will discuss in Chapter 17), looking for matching records. Even if the query could be performed without excess searching (for example, via the index join of Chapter 21), the disk drive will still have to seek repeatedly as it alternates between reading the STUDENT and DEPT blocks.

A non-homogeneous organization would store the STUDENT and DEPT records in the same file, such that the record for each student will be stored near the record for its major department. Figure 15-2 depicts the first two blocks of such an organization, assuming 3 records per block. The file consists of a DEPT record, followed by the STUDENT records having that department as a major. This organization requires fewer block accesses to calculate the join, because the joined records are *clustered* on the same (or a nearby) block.

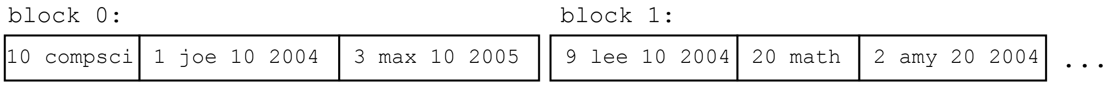


Figure 15-2: Clustered, non-homogeneous records

Clustering improves the efficiency of queries that join the clustered tables, because the matching records are stored together. However, clustering will cause single-table queries to become less efficient, because the records for each table are spread out over more blocks. Similarly, joins with other tables will also be less efficient. Thus clustering is effective only if the most heavily-used queries perform the join encoded by the clustering.[†]

15.1.3 Fixed-length vs. variable-length fields

Every field in a table has a defined type. Based on that type, the record manager decides whether to implement the field using a *fixed-length* or *variable-length* representation. A fixed-length representation uses exactly the same number of bytes to store each of its values, whereas a variable-length representation expands and contracts based on the data value stored.

A fixed-length field representation uses the same number of bytes to hold each value of the field.

[†] In fact, clustering is the fundamental organizational principle behind the early hierarchical database systems, such as IBM's *IMS* system. Databases that are naturally understood hierarchically can be implemented very efficiently in such systems.

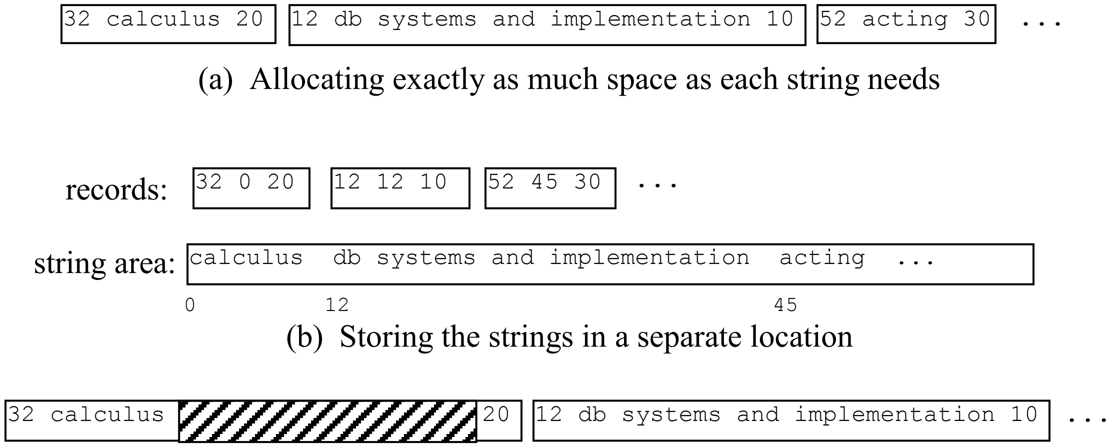
Most types are naturally fixed-length. For example, both integers and floating-point numbers can be stored as 4-byte binary values. In fact, all numeric and date/time types have natural fixed-length representations. The Java type *String* is the prime example of a type that needs a variable-length representation, because character strings can be arbitrarily long.

Variable-length representations are less desirable than fixed-length representations because they can cause significant complications. Consider for example a record sitting in the middle of a block that is packed with records, and suppose that one of its field values gets modified. If the field is fixed-length, then the record will remain the same size, and the field can be modified in place. However, if the field is variable-length, the record may get larger. In order to make room for the larger record, the record manager may have to rearrange the location of the records in the block. In fact, if the modified record gets too large, then one or more records might need to be moved out of the block and placed in a different block.

Consequently, the record manager tries its best to use a fixed-length representation whenever possible. For example, the record manager can choose from three different representations of a string field:

- A variable-length representation, in which the record manager allocates the exact amount of space in the record that the string requires.
- A fixed-length representation, in which the record manager stores the string in a location outside of the record, and stores in the field a fixed-length reference to that location.
- A fixed-length representation, in which the record manager allocates the same amount of space in the record for each string, regardless of its length.

These representations are depicted in Figure 15-3, for some of the records in the COURSE table.


32 calculus 20
12 db systems and implementation 10
52 acting 30
...
records:
32 0 20
12 12 10
52 45 30
...
string area:
calculus db systems and implementation acting ...
0
12
45
32 calculus
20
12 db systems and implementation 10
...

(a) Allocating exactly as much space as each string needs

(b) Storing the strings in a separate location

(c) Allocating the same amount of space for each string

Figure 15-3: Alternative representations for the *Title* field in COURSE records

The first representation creates variable-length records; these records are space-efficient, but have the problems that we just discussed. The other two representations are fixed-length alternatives.

The second representation moves the strings out of the record and into a separate "string area". This area could be a separate file, or (if the strings are very large) a directory in which each string is stored in its own file. In either case, the field contains a reference to the string's location in that area. This representation results in records that are both fixed-length and small. Small records are good, because they can be stored in fewer blocks and thus require fewer block accesses. The downside to this representation is that the record is fragmented, which means that retrieving the string value from a record requires an additional block access.

The third representation has the advantage that the records are fixed-length and the strings are stored in the record. However, the downside is that most records will be larger than they need to be. If there is a wide difference in string sizes, then this wasted space will be significant, resulting in a larger file and correspondingly more block accesses.

None of these representations are clearly better than the others. As a way to help the record manager choose the proper representation, standard SQL provides three different string datatypes: *char*, *varchar*, and *clob*. We encountered the *char* and *varchar* types in Section 4.3.2: the type *char(n)* specifies strings of exactly *n* characters, whereas the type *varchar(n)* specifies strings of at most *n* characters. The type *clob(n)* also specifies strings of at most *n* characters. The difference between *varchar(n)* and *clob(n)* is the expected size of n. In *varchar(n)*, n is reasonably small, say no more than 4K. On the other hand, the value of *n* in *clob(n)* can be in the giga-character range. (The acronym CLOB stands for "character large object".) For an example of a *clob* field, suppose that we add a field *Syllabus* to the SECTION table of our university database, with the idea that the values of this field would contain the text of each section's syllabus. If we assume that syllabi can be no more than 8000 characters, then we could reasonably define the field as *clob(8000)*.

Fields of type *char* most naturally correspond to the third representation. Since all strings will be the same length, there is no wasted space inside the records, and the fixed-length representation will be most efficient.

Fields of type *varchar(n)* most naturally correspond to the first representation. Since SQL requires that *n* be relatively small, we know that placing the string inside the record will not make the record too large. Moreover, the possibly large variance in string sizes means that the fixed-length representation would waste too much space. Thus, the variable-length representation is the best alternative.

If *n* happens to be very small (say, less than 20), then the record manager might choose to implement a *varchar* field using the third representation. The reason is that there will not be that much wasted space, and so the benefits of a fixed-length representation become more important.

Fields of type *clob* correspond to the second representation, because that representation handles large strings the best. The other representations place the strings directly inside the records, which leads to exceptionally large records and an exceptionally large data file. By storing the large string outside of the record, the records become smaller, and more manageable.

15.1.4 Placing fields in records

The record manager must also determine the structure of its records. For fixed-length records, it needs to compute the following values:

- the size of each record in a file; and
- the offset within the record of each field.

The most straightforward strategy is to determine the size of each field, and to store the fields next to each other in the record. The size of the record then becomes the sum of the sizes of the fields, and the offset of a field becomes the location where the previous field ends.

The size of a fixed-length field is determined by its type. For example:

- Integers require 4 bytes.
- A fixed-length representation of *char(n)* and *varchar(n)* strings requires 4 + n*c bytes, where c is the number of bytes per character. The additional 4 bytes is for an integer that holds the value n. Other string implementations will have different sizes. For example, Exercise 12.18 asked you to represent a string using a delimiter character instead of an integer; in that case, a string requires 2*(n+1) bytes.

This strategy of tightly packing fields into records works for Java-based systems (like SimpleDB and Derby), but can cause problems elsewhere. The issue has to do with ensuring that values are aligned properly in memory. In most computers, the machine code to access an integer requires that the integer be stored in a memory location that is a multiple of 4; we say that the integer must be *aligned* on a 4-byte boundary. The record manager must therefore ensure that every integer in every page is aligned on a 4-byte boundary. Since OS pages are always aligned on a 2^N-byte boundary for some reasonably-large N, the first byte of each page will be properly aligned. Thus the record manager must simply make sure that the offset of each integer within each page is a multiple of 4. If the previous field ended at a location that is not a multiple of 4, then the record manager must *pad* it with enough bytes so that it does.

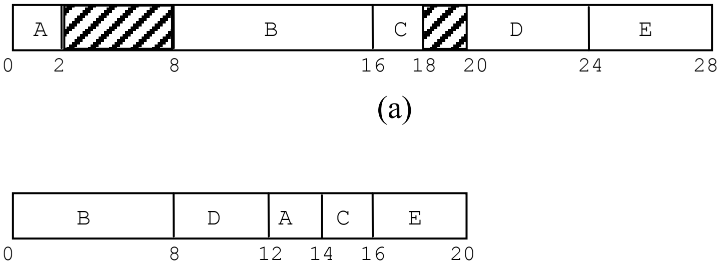
> *The record manager pads fields so that*
> *each field is aligned on an appropriate byte boundary.*

For example, consider the STUDENT table, which consists of three integer fields and a varchar(10) string field. The integer fields are multiples of 4, so they don't need to be padded. The string field, however, requires 14 bytes (assuming 1 byte per character); it therefore needs to be padded with 2 additional bytes, so that the field following it will be aligned on a multiple of 4.

In general, different types may require different amounts of padding. Double-precision floating point numbers, for example, are usually aligned on an 8-byte boundary, and small integers are usually aligned on a 2-byte boundary. The record manager is responsible for ensuring these alignments. A simple strategy is to position the fields in the order that they were declared, padding each field to ensure the proper alignment of the next field. A more clever strategy is to reorder the fields so that the least amount of padding is required. For example, consider the following SQL table declaration:

```
create table T (A smallint, B double precision, C smallint, D int, E int)
```

Suppose the fields are stored in the order given. Then field A needs to be padded with 6 extra bytes, and field C needs to be padded with 2 extra bytes, leading to a record length of 28 bytes; see Figure 15-4(a). On the other hand, if the fields are stored in the order [B, D, A, C, E], then no padding is required and the record length is only 20 bytes, as shown in Figure 15-4(b).



(a)

(b)

Figure 15-4: Placing fields in a record to establish alignment

In addition to padding fields, the record manager must also pad each record. The idea is that each record needs to end on a k-byte boundary, where k is the largest supported alignment, so that every record in a page has the same alignment as the first one. Consider again the field placement of Figure 15-4(a), which has a record length of 28 bytes. Suppose that the first record begins at byte 0 of the block. Then the second record will start at byte 28 of the block, which means that field B of the second record will start at byte 36 of the block, which is the wrong alignment. It is essential that each record begin on an 8-byte boundary. In the example of Figure 15-4, the records of both part (a) and part (b) need to be padded with 4 additional bytes.

A Java program does not need to consider padding because it cannot directly access numeric values in a byte array. For example, the Java method to read an integer from a page is *ByteBuffer.getInt*. This method does not call a machine-code instruction to obtain the integer, but instead constructs the integer itself from the 4 specified bytes of the array.

This activity is less efficient than a single machine-code instruction, but it avoids alignment issues.[†]

15.2 Implementing a File of Records

Now that we have considered the various decisions that the record manager must address, we turn to how these decisions get implemented. We shall begin with the most straightforward implementation: a file containing homogeneous, unspanned, and fixed-length records. We then consider how this implementation would change if other decisions were made.

15.2.1 A straightforward implementation

Suppose we want to create a file of homogeneous, unspanned, and fixed-length records. The fact that the records are unspanned means that we can treat the file as a sequence of blocks, where each block contains its own set of records. The fact that the records are homogeneous and fixed-length means that we can allocate the same amount of space for each record within a block. In other words, we can think of each block as an *array of records*. We call such a block a *record page*.

> *A record page is a block that is structured as an array of fixed-length records.*

The record manager can implement record pages as follows. It divides a block into *slots*, where each slot is large enough to hold a record plus one additional byte. The value of this byte is a flag that denotes whether the slot is empty or in use; let's say that a 0 means "empty" and a 1 means "in use"[†].

For example, suppose that the block size is 400 and the record size is 26; then each slot is 27 bytes long, and the block holds 14 slots with 22 bytes of wasted space. Figure 15-5 depicts this situation. This figure shows 4 of the 14 slots; slots 0 and 13 currently contain records, whereas slots 1 and 2 are empty.

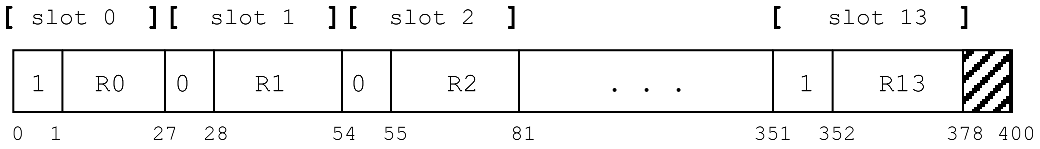


Figure 15-5: A record page with space for fourteen 26-byte records

[†] The Java approach also has the advantage of being machine-independent. Computers can differ on which order they store the bytes of an integer, leading to conflicts between "big-endian" and "little-endian" machines. (If you create a database on a little-endian machine, you may not be able to read it on a big-endian machine because numbers will be interpreted incorrectly.) By handling the byte-to-integer conversion itself, Java avoids these conflicts.

[†] We can improve the space usage by only using a bit to hold each empty/inuse flag. See Exercise 15.8.

The record manager needs to be able to insert, delete, and modify records in a record page. In order to do so, it uses the following information about the records:

- the record length;
- the name, type, length, and offset of each field of a record.

We call these values the *table information*.

For example, consider the table STUDENT, as defined at the end of Chapter 2. A STUDENT record contains three integers plus a 10-character *varchar* field. If we use the storage strategy of SimpleDB, then each integer requires 4 bytes and a 10-character string requires 14 bytes. Let's also assume that padding is not necessary, and that *varchar* fields are implemented by allocating fixed space for the largest possible string. Then Figure 15-6 gives the table information for this table. Note that the length information for *SName* is the length of the field in bytes, not the maximum number of characters in the field.

`Record Length: 26`

`Field Information:`

Name	**Type**	**Length**	**Offset**
SId	int	4	0
SName	varchar	14	4
GradYear	int	4	18
MajorId	int	4	22

Figure 15-6: The table information for STUDENT

This table information allows the record manager to determine where values are located within the block. For example:

- The empty/inuse flag for slot k is at location `(RL+1)*k`, where RL is the record length.
- The value of field F in the record at slot k is at location `(RL+1)*k + OFF + 1`, where RL is the record length and OFF is the offset of the field in the record.

The record manager can therefore process insertions, deletions, and modifications quite easily:

- To insert a new record, the record manager examines the empty/inuse flag of each slot until it finds a 0. It then sets the flag to 1, and returns the slot number. If all flag values are 1, then the block is full and insertion is not possible.
- To delete the value of the record in slot k, the record manager simply sets the empty/inuse flag at that slot to 0.
- To modify a field value of the record at slot k, (or to initialize a field of a new record), the record manager determines the location of that field, and writes the value to that location.

The record manager can also perform retrievals just as easily:

- To iterate through all records in the block, the record manager examines the empty/inuse flag of each slot. Each time it finds a 1, it knows that that slot contains an existent record.
- To retrieve the value of a field from the record at slot k, the record manager determines the location of that field, and reads the value from that location.

As you can see from these operations, the record manager needs a way to identify a record. When records are fixed-length, the most straightforward record identifier is its slot number.

> *Each record in a page has an ID.*
> *When records are fixed-length, the ID can be its slot number in the page.*

15.2.2 Implementing Variable-Length Fields

The implementation of fixed-length fields is very straightforward. In this section we consider how this implementation changes when variable-length fields are added to the mix.

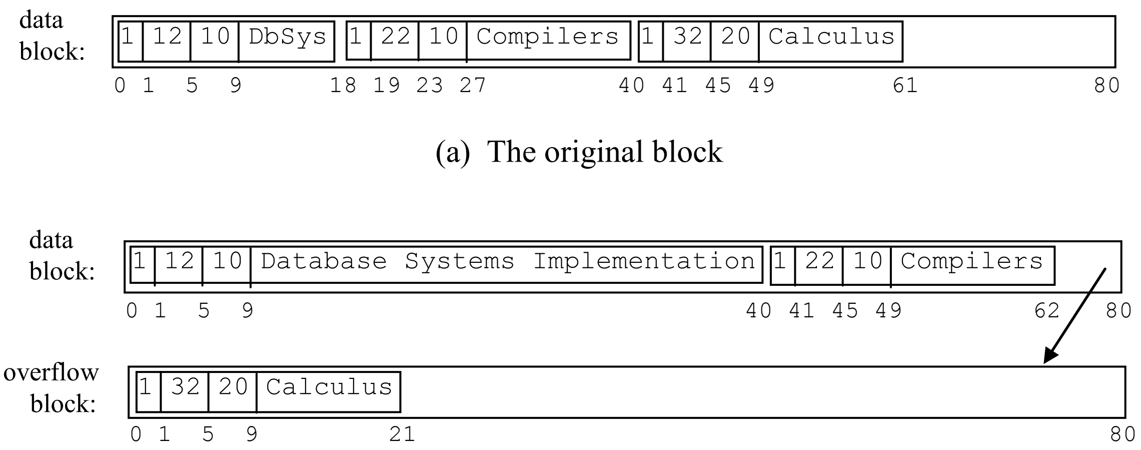
The first thing we notice is that the field offsets in a record are no longer fixed. In particular, the offsets of all of the fields following a variable-length field will differ from record to record. The only way to determine the offset of those fields is to read the previous field and see where it ends. If the first field in a record is variable-length, then it will be necessary to read the first n-1 fields of the record in order to determine the offset of the nth field. Consequently, the record manager typically places the fixed-length fields at the beginning of each record, so that they can be accessed by a precomputed offset. The variable-length fields are placed at the end of the record. The first variable-length field will have a fixed offset, but the remaining ones will not.

The next thing we notice is that the records can have different lengths. This has two important consequences.

- *It is no longer possible to move to a particular record by multiplying its slot number by the slot size.* The only way to find the beginning of a record is to read to the end of the previous record.
- *Modifying a field value can cause a record's length to change.* In this case, the block contents to the right of the modified value must be shifted to the right (if the new value is larger) or the left (if the new value is smaller). If we are unlucky, the shifted records will spill out of the block; this situation must be handled by allocating an *overflow block*.

An overflow block is a new block allocated from an area known as the *overflow area*. Any record that spills out of the original block is removed from that block and added to an overflow block. If we are really unlucky, very many fields will get modified to a larger value, and a chain of several overflow blocks may be necessary. Each block will contain a reference to the next overflow block on the chain. Conceptually, the original and overflow blocks form a single (large) record page.

For example, consider the COURSE table, and suppose that course titles are saved as variable-length strings. Figure 15-7(a) depicts a block containing the first three records of the table. (The *Title* field has been moved to the end of the record, because the other fields are fixed-length.) Figure 15-7(b) depicts the result of modifying the title "DbSys" to "Database Systems Implementation". Assuming a block size of 80 bytes, the third record no longer fits in the block, and so it is placed in an overflow block. The original block contains a reference to that overflow block.


data block:
1 | 12 | 10 | DbSys | 1 | 22 | 10 | Compilers | 1 | 32 | 20 | Calculus
0 1 5 9 18 19 23 27 40 41 45 49 61 80
data block:
1 | 12 | 10 | Database Systems Implementation | 1 | 22 | 10 | Compilers
0 1 5 9 40 41 45 49 62 80
overflow block:
1 | 32 | 20 | Calculus
0 1 5 9 21 80

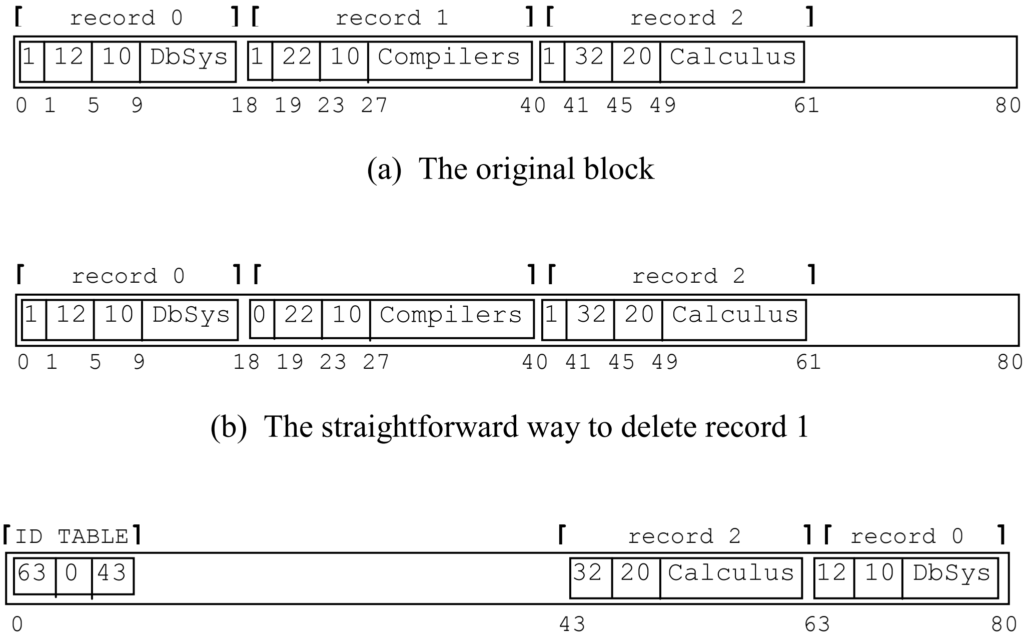
(a) The original block

(b) The result of modifying the title of course 12

Figure 15-7: Using an overflow block to implement variable-length records

Apart from the need for overflow blocks, the complications introduced by variable-length fields do not impact overall efficiency. Granted, the record manager now has to do more: It shifts records every time a field is modified, and it must completely read each record in order to move to the next one. However, this activity does not require any additional block accesses, and thus is not overly time-consuming. Nevertheless, there is one improvement that is often made.

Consider record deletion. Figure 15-8(a) depicts a block containing the first three COURSE records, the same as in Figure 15-7(a). Deleting the record for course 22 sets the flag to 0 (for "empty"), but leaves the record intact, as shown in Figure 15-8(b). Because a new record will not necessarily fit into the empty space left by this record, we would like to reclaim the space by physically removing the record from the block and shifting the records after it to the left. However there is a problem: the record manager uses the record's slot as its ID, and so eliminating the deleted record will cause the subsequent records in the block to have the wrong ID.



(a) The original block

(b) The straightforward way to delete record 1

(c) A better way to delete record 1

Figure 15-8: Using an ID table to implement variable-length records

The solution to this problem is to dissociate the record's ID from its slot. That is, we add an array of integers to the block, called its *ID-table*. Each entry in the array denotes an ID. The value of an entry is the offset of the record having that ID; if there is no such record, the entry has the value 0. Figure 15-8(c) depicts the same data as 15-8(b), but with an ID-table. The ID table contains three entries: two of them point to the records at offsets 63 and 43 of the block, and the other is empty.

The ID-table provides a level of indirection that allows the record manager to move records around inside of a block. If the record moves, its entry in the ID-table is adjusted correspondingly; and if the record is deleted, its entry is set to 0. When a new record is inserted, the record manager finds an available entry in the table and assigns it as the ID of the new record.

An ID-table allows variable-length records to be moved within a block, while providing each record with a fixed identifier.

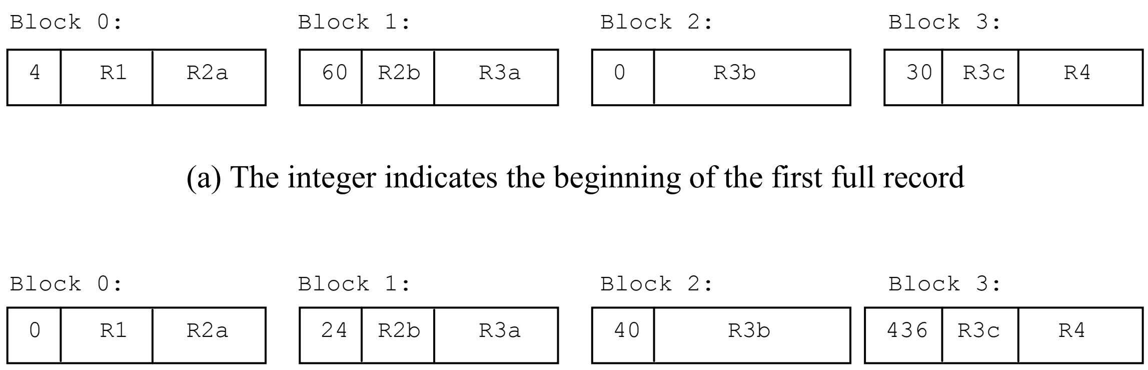
Note that the ID-table expands as the number of records in the block increases. The size of the array is necessarily open-ended, because a block can hold a varying number of variable-length records. Typically the ID-table is placed at one end of the block and the records are placed at the other end, and they grow towards each other. This situation can be seen in Figure 15-8(c), where the first record in the block is at its far right.

Also note that with an ID-table, there is no need for empty/inuse flags. A record is in use if there is an entry of the ID-table pointing to it. Empty records have an ID of 0 (and in fact don't even exist). The ID-table also helps the record manager find each record in the block quickly. To move to a record having a particular ID, the record manager simply uses the offset stored in that entry of the ID-table; and to move to the next record, the record manager scans the ID-table until it finds the next non-zero entry.

15.2.3 Implementing Spanned Records

We next consider how our implementation can be modified to handle spanned records.

When records are unspanned, the first record in each block always begins at the same location. With spanned records, this situation is no longer true. Consequently, the record manager must store information in each block to help it determine where the records are. There are several possible implementation strategies. One strategy is for the record manager to store an integer at the beginning of each block; that integer would hold the offset of the first record. For example, consider Figure 15-9(a). The first integer in block 0 is a 4, denoting that the first record R1 begins at offset 4 (that is, immediately after the integer). Record R2 spans blocks 0 and 1, and so the first record in block 1 is R3, which begins at offset 60. Record R3 continues through block 2 into block 3. Record R4 is the first record in block 3, and begins at offset 30. Note that the first integer of block 2 is 0, denoting the fact that no record begins in that block.



(a) The integer indicates the beginning of the first full record

(b) The integer indicates the omitted portion of the first record

Figure 15-9: Two ways to implement spanned records

A second strategy would be to keep an integer at the beginning of each block, but to interpret it differently. Namely, the integer would denote where the first record actually begins. If the first record is not spanned, then the integer would be 0. Otherwise, the integer is the number of bytes of the record that appear in earlier blocks. For example, consider Figure 15-9(b). The value of 0 in block 0 indicates that R1 is not spanned. The value of 24 in block 1 indicates that the first 24 bytes of R2 are in an earlier block. The value of 436 in block 3 indicates that the first 436 bytes of R3 are in earlier blocks. Here we assume a block size of 400 bytes; since the record manager knows the block size, it can calculate how many blocks the record has consumed so far.

In addition to these types of implementation decisions, the record manager can choose to split a spanned record in two different ways. The first way is to fill the block as much as possible, splitting it on the block boundary; the remaining bytes are placed into the next block(s) of the file. The second way is to write the record value-by-value; when the page becomes full, the writing continues on a new page. The first way has the advantage that it wastes absolutely no space, but has the disadvantage of splitting a value across blocks. To access the split value, the record manager must reconstruct the value by catenating the bytes from the two blocks.

15.2.4 Implementing Non-Homogeneous Records

If the record manager supports non-homogeneous records, then it will also need to support variable-length records, because records from different tables need not be the same size.

There are two issues related to having non-homogeneous records in a block:

- The record manager needs to know the table information for each type of record in the block.
- Given a record, the record manager needs to know which table it comes from.

The record manager can address the first issue by keeping an array of table-information values, one for each possible table. The record manager can address the second issue by adding an extra value to each record slot; this value, sometimes called a *tag value*, is an index into the table-information array, which specifies the table that the record belongs to.

For example, consider again Figure 15-2, which depicts non-homogeneous blocks from the DEPT and STUDENT tables. The record manager will keep an array containing the table information from both of these tables; let’s assume that the DEPT information is in index 0 of the array, and the STUDENT information is in index 1. Then the slot for each record from DEPT will contain a tag value of 0, and the slot for each STUDENT record will contain a tag value of 1.

The behavior of the record manager does not need to change very much. When the record manager accesses a record, it determines from the tag value which table information to use. It can then use that table to read or write to any field, the same as in the homogeneous case.

The log records in SimpleDB are an example of non-homogeneous records. The first value of each log record is an integer that indicates the type of the log record. The recovery manager uses that value to determine how to read in the rest of the record.

15.3 The SimpleDB Record Manager

Now that we seen conceptually how the record manager works, we can get down to specifics. This section examines the SimpleDB record manager, which implements the basic record manager of Section 15.2.1. Some of the end-of-chapter exercises ask you to extend it with the variations we have discussed.

15.3.1 Managing table information

The SimpleDB record manager uses the classes *Schema* and *TableInfo* manage a record's table information. Their API appears in Figure 15-10.

Schema

```
public Schema();

public void addField(String fldname, int type, int length);
public void addIntField(String fldname);
public void addStringField(String fldname, int length);
public void add(String fldname, Schema sch);
public void addAll(Schema sch);

public Collection<String> fields();
public boolean hasField(String fldname);
public int type(String fldname);
public int length(String fldname);
```

TableInfo

```
public TableInfo(String tblname, Schema schema);
public TableInfo(String tblname, Schema schema,
                 Map<String,Integer> offsets, int recordlen);

public String fileName();
public Schema schema();
public int offset(String fldname);
public int recordLength();
```

Figure 15-10: The API for SimpleDB table information

A *Schema* object holds the schema of a table – namely, the name and type of each field, and the length of each string field. This information corresponds to what a user would specify when creating a table, and contains no physical knowledge. For example, the length of a string is the maximum number of characters allowed, not its size in bytes.

A schema can be thought of as a collection of triples of the form [fieldname, type, length]. The class *Schema* contains five methods to add a triple to the collection. The method *addField* adds a triple explicitly. The methods *addIntField*, *addStringField*, *add*, and *addAll* are convenience methods; the first two of these methods calculate the triple for the client, and the last two copy triples from an existing schema. The class also has accessor methods to retrieve the collection of all field names, determine if a specified field is in the collection, and retrieve the type and length of a specified field.

The class *TableInfo* contains the physical information about a table, as well as its schema. It calculates field and record sizes, the field offsets within the record, and the name of the file where the records will be stored. The class has two constructors, corresponding to the two reasons for creating a *TableInfo* object. The first constructor is called when a table is created; it takes the table's name and schema as arguments and calculates the physical size and offset of each field. The second constructor is called after retrieving the information for a previously-created table; the caller simply provides the previously-calculated values. The class also contains accessor methods for returning the filename, the length of the record, and offset of each field.

Figure 15-11 contains sample code illustrating the use of these two classes. The first part of the code creates a schema containing the three fields of the COURSE table, and then creates a *TableInfo* object from it. The second part of the code iterates through every field in the schema, printing its field name and offset.

```
Schema sch = new Schema();
sch.addIntField("cid");
sch.addStringField("title", 20);
sch.addIntField("deptid");
TableInfo ti = new TableInfo("course", sch);

for (String fldname : ti.schema().fields()) {
      int offset = ti.offset(fldname);
      System.out.println(fldname + " has offset " + offset);
}
```

Figure 15-11: Specifying the structure of COURSE records

15.3.2 Implementing *Schema* and *TableInfo*

The code for the class *Schema* is straightforward, and appears in Figure 15-12.

```
public class Schema {
   private Map<String,FieldInfo> info =
                              new HashMap<String,FieldInfo>();

   public void addField(String fldname, int type, int length) {
      info.put(fldname, new FieldInfo(type, length));
   }

   public void addIntField(String fldname) {
      addField(fldname, INTEGER, 0);
   }

   public void addStringField(String fldname, int length) {
      addField(fldname, VARCHAR, length);
   }

   public void add(String fldname, Schema sch) {
      int type   = sch.type(fldname);
      int length = sch.length(fldname);
      addField(fldname, type, length);
   }

   public void addAll(Schema sch) {
      info.putAll(sch.info);
   }

   public Collection<String> fields() {
      return info.keySet();
   }

   public boolean hasField(String fldname) {
      return fields().contains(fldname);
   }

   public int type(String fldname) {
      return info.get(fldname).type;
   }

   public int length(String fldname) {
      return info.get(fldname).length;
   }

   class FieldInfo {
      int type, length;
      public FieldInfo(int type, int length) {
         this.type = type;
         this.length = length;
      }
   }
}
```

Figure 15-12: The code for SimpleDB class *Schema*

Internally, the class stores each triple in a map keyed on the field name. The length and type are the mapped values. The private class *FieldInfo* combines the length and type into a single object.

Types are denoted by the constants INTEGER and VARCHAR, as defined in the JDBC class *Types*. The length of a field is only meaningful for string fields; the method *addIntField* gives integers a length value of 0, but this value is irrelevant as it will never be accessed.

The code for the class *TableInfo* appears in Figure 15-13. The first constructor calculates the offsets of each field; the fields are positioned in the order they appear in the hash map, which is essentially random. The constructor also calculates the length of each field in bytes, the record length as the sum of the field lengths, and assigns the offset of each field to be the location at which the previous field ends. In other words, no padding is performed.

```
public class TableInfo {
   private Schema schema;
   private Map<String,Integer> offsets;
   private int recordlen;
   private String tblname;

   public TableInfo(String tblname, Schema schema) {
      this.schema  = schema;
      this.tblname = tblname;
      offsets = new HashMap<String,Integer>();
      int pos = 0;
      for (String fldname : schema.fields()) {
         offsets.put(fldname, pos);
         pos += lengthInBytes(fldname);
      }
      recordlen = pos;
   }

   public TableInfo(String tblname, Schema schema,
                    Map<String,Integer> offsets, int recordlen) {
      this.tblname   = tblname;
      this.schema    = schema;
      this.offsets   = offsets;
      this.recordlen = recordlen;
   }

   public String fileName() {
      return tblname + ".tbl";
   }

   public Schema schema() {
      return schema;
   }

   public int offset(String fldname) {
      return offsets.get(fldname);
   }

   public int recordLength() {
      return recordlen;
   }

   private int lengthInBytes(String fldname) {
      int fldtype = schema.type(fldname);
      if (fldtype == INTEGER)
         return INT_SIZE;
      else
         return STR_SIZE(schema.length(fldname));
   }
}
```

Figure 15-13: The code for the SimpleDB class *TableInfo*

15.3.3 Managing the Records in a Page

The SimpleDB class *RecordPage* manages the records within a page. Its API appears in Figure 15-14.

RecordPage

```
public RecordPage(Block blk, TableInfo ti, Transaction tx);

public boolean next();
public int     getInt(String fldname);
public String  getString(String fldname);
public void    setInt(String fldname, int val);
public void    setString(String fldname, String val);
public void    close();

public void    delete();
public boolean insert();

public void    moveToId(int id);
public int     currentId();
```

Figure 15-14: The API for SimpleDB record pages

Several methods in this API are similar to those of the JDBC *ResultSet* interface. In particular, each *RecordPage* object keeps track of a "current record". The method *next* moves to the next record in the page, returning *false* if there are no more records. The *get/set* methods access the value of a specified field in the current record, and the *delete* method deletes the current record. As in JDBC, the deleted record remains current; the client code must call *next* to move to the next record.

The *insert* method does not operate on the current record. Instead, it inserts a blank record somewhere in the page, returning *false* if there is no space for a new record. The new record becomes current.

The *RecordPage* constructor takes three arguments: the block to be accessed, the *TableInfo* object that describes the records in the block, and a *Transaction* object. The record page uses the *Transaction* object to pin and unpin the block, and to get/set values in the page.

Every record in a page has an identifier that is assigned to it when it is inserted, and stays with it until it is deleted. Identifiers are important for implementing indexing, as we shall see in Chapter 21. Class *RecordPage* provides two methods that interact with identifiers: Method *moveToId* sets the current record to be the one having the specified identifier; and method *currentId* returns the identifier of the current record.

Figure 15-15 continues the example code of Figure 15-11, examining the records in block 337 of the COURSE table. Part 1 of the code modifies any course titled "VB programming" to have the title "Java programming", and deletes the courses offered by department 30. Part 2 inserts a new record into the block (if possible), having a course ID of 82, title "OO Design" and department 20.

```
SimpleDB.init("studentdb");
TableInfo ti = ... //info for the COURSE table, as in Fig 15-11

Transaction tx = new Transaction();
String filename = ti.fileName();
Block blk = new Block(filename, 337);
RecordPage rp = new RecordPage(blk, ti, tx);

// Part 1
while (rp.next()) {
      int dept = rp.getInt("deptid");
      if (dept == 30)
            rp.delete();
      else if (rp.getString("title").equals("VB programming"))
            rp.setString("title", "Java programming");
}

// Part 2
boolean ok = rp.insert();
if (ok) {
      rp.setInt("cid", 82);
      rp.setString("title", "OO Design");
      rp.setInt("deptid", 20);
}
rp.close();
tx.commit();
```

Figure 15-15: Accessing the records in a page

15.3.4 Implementing record pages

SimpleDB implements the slotted-page structure of Figure 15-5. The only difference is that the empty/inuse flags are implemented as 4-byte integers instead of single bytes (the reason being that SimpleDB doesn't support byte-sized values). The code for the class *RecordPage* appears in Figure 15-16.

```
public class RecordPage {
   public static final int EMPTY = 0, INUSE = 1;

   private Block blk;
   private TableInfo ti;
   private Transaction tx;
   private int slotsize;
   private int currentslot = -1;

   public RecordPage(Block blk, TableInfo ti, Transaction tx) {
      this.blk = blk;
      this.ti = ti;
      this.tx = tx;
      tx.pin(blk);
      slotsize = ti.recordLength() + INT_SIZE;
   }

   public void close() {
      if (blk != null) tx.unpin(blk);
      blk = null;
   }

   public boolean next() {
      return searchFor(INUSE);
   }

   public int getInt(String fldname) {
      int position = fieldpos(fldname);
      return tx.getInt(blk, position);
   }

   public String getString(String fldname) {
      int position = fieldpos(fldname);
      return tx.getString(blk, position);
   }

   public void setInt(String fldname, int val) {
      int position = fieldpos(fldname);
      tx.setInt(blk, position, val);
   }

   public void setString(String fldname, String val) {
      int position = fieldpos(fldname);
      tx.setString(blk, position, val);
   }

   public void delete() {
      int position = currentpos();
      tx.setInt(blk, position, EMPTY);
   }

   public boolean insert() {
      currentslot = -1;
      boolean found = searchFor(EMPTY);
      if (found) {
         int position = currentpos();
         tx.setInt(blk, position, INUSE);
      }
      return found;
   }

   public void moveToId(int id) {
      currentslot = id;
   }

   public int currentId() {
      return currentslot;
   }

   private int currentpos() {
      return currentslot * slotsize;
   }

   private int fieldpos(String fldname) {
      int offset = INT_SIZE + ti.offset(fldname);
      return currentpos() + offset;
   }

   private boolean isValidSlot() {
      return currentpos() + slotsize <= BLOCK_SIZE;
   }

   private boolean searchFor(int flag) {
      currentslot++;
      while (isValidSlot()) {
         int position = currentpos();
         if (tx.getInt(blk, position) == flag)
            return true;
         currentslot++;
      }
      return false;
   }
}
```

Figure 15-16: The code for the SimpleDB class *RecordPage*

The constructor initializes the current slot to be -1; that is, it is one slot before the first slot in the page. The methods *next* and *insert* call the private method *searchFor* to find a slot having the specified flag; the method *next* searches for INUSE, whereas *insert* searches for EMPTY. Method *searchFor* repeatedly increments the current slot until it either finds a slot having the specified flag or it runs out of slots.

The *get/set* methods call the corresponding methods in *Transaction*. The difference between these methods is their arguments: The *Transaction* methods are given an offset with respect to the beginning of the page, whereas the *RecordPage* methods are given a field name. The *RecordPage* methods are thus responsible for calculating the page offset for the given fieldname. This offset is the position of the current record, plus 4 bytes for the flag, plus the offset of the field within the record.

The *delete* method simply sets the flag of the current record to EMPTY. The *close* method unpins the buffer. It is therefore crucial that clients close record pages when no longer needed.

15.3.5 Formatting record pages

A record page has a specific structure – namely, it is partitioned into slots, with the value of the first integer in each slot indicating whether that record exits. A record page therefore needs to be formatted correctly before it can be used. The class *RecordFormatter* performs this service, via its method *format*; see Figure 15-17.

```
class RecordFormatter implements PageFormatter {
   private TableInfo ti;

   public RecordFormatter(TableInfo ti) {
      this.ti = ti;
   }

   public void format(Page page) {
      int recsize = ti.recordLength() + INT_SIZE;
      for (int pos=0; pos+recsize<=BLOCK_SIZE; pos += recsize) {
         page.setInt(pos, EMPTY);
         makeDefaultRecord(page, pos);
      }
   }

   private void makeDefaultRecord(Page page, int pos) {
      for (String fldname : ti.schema().fields()) {
         int offset = ti.offset(fldname);
         if (ti.schema().type(fldname) == INTEGER)
            page.setInt(pos + INT_SIZE + offset, 0);
         else
            page.setString(pos + INT_SIZE + offset, "");
      }
   }
}
```

Figure 15-17: The code for the SimpleDB class *RecordFormatter*

The *format* method loops through the slots of the page. For each slot, the method sets the flag to EMPTY, and initializes every field to a default value: Integers are set to 0 and strings are set to `""`. Note that the formatter needs to know the appropriate *TableInfo* object, so that it can determine the size of each slot and the offset and type of each field.

A *RecordFormatter* object is needed whenever a new block gets appended to a file. Figure 15-18 contains some example code demonstrating it use. The code appends a new block to the COURSE table, and then inserts a new record into it, having default values. Note that the call to *insert* will set the flag of slot 0 to INUSE, but will not change the contents of the slot.

```
SimpleDB.init("studentdb");
TableInfo ti = ... //info for the COURSE table, as in Figure 15-11

Transaction tx = new Transaction();
RecordFormatter fmtr = new RecordFormatter(ti);
Block blk = tx.append(ti.fileName(), fmtr);
RecordPage rp = new RecordPage(blk, ti, tx);
rp.insert();
rp.close();
tx.commit();
```

Figure 15-18: Appending a new block to a file

The formatter is passed into the *Transaction.append* method. That method passes it to *Buffermgr.pinNew*, which passes it to *Buffer.assignToNew*. This last method (finally!) calls the formatter's *format* method to format the page before writing it to disk.

15.3.6 Managing the Records in a File

A record page can hold only a relatively small number of records. A *record file* holds a large number of similar records in a file; its API is given in Figure 15-19.

RecordFile

```
public RecordFile(TableInfo ti, Transaction tx);

// methods that establish the current record
public void    beforeFirst();
public boolean next();
public void    moveToRid(RID r);
public void    insert();
public void    close();

// methods that access the current record
public int     getInt(String fldname);
public String  getString(String fldname);
public void    setInt(String fldname, int val);
public void    setString(String fldname, String val);
public RID     currentRid();
public void    delete();
```

RID

```
public RID(int blknum, int id);
public int     blockNumber();
public int     id();
```

Figure 15-19: The API for SimpleDB record files

As with a record page, a record file keeps track of the current record, and its API contains methods to change the current record and to access its contents. The method *next* positions the current record to the next record in the file. If the current block has no more records, then the succeeding blocks in the file will be read until another record is found. Method *beforeFirst* positions the current record to be before the first record of the file, and is used by clients that need to repeatedly scan the file.

The *get/set* and *delete* methods apply to the current record. The *insert* method inserts a new record somewhere in the file, starting with the current record's block. Unlike with *RecordPage*, this insertion method always succeeds; if it cannot find a place to insert the record in the existing blocks of the file, it appends a new block to the file and inserts the record there.

A record in a file can be identified by a pair of values: its logical block reference and its identifier within the block. These two values are known as a *rid* (which stands for *record identifier*). Record identifiers are implemented in the class *RID*. The class constructor saves the two values; the accessor methods *block* and *id* retrieve them.

Class *RecordFile* contains two methods that interact with rids. Method *moveToRid* positions the current record at the specified rid; it assumes that the rid denotes an existing record in the file. And method *currentRid* returns the rid of the current record.

The *RecordFile* class provides a level of abstraction significantly different from the other classes we have seen so far. That is, the methods of *Page*, *Buffer*, *Transaction*, and *RecordPage* all apply to a particular block. The *RecordFile* class, on the other hand, hides the block structure from its clients. In general, a client will not know (or care) which block is currently being accessed. It only needs to remember to close the record file when it is done.

The code of Figure 15-20 illustrates the use of record files. This code assumes that the records in file *junk* each contain a single integer field named *A*. The code inserts 10,000 more records into the file, each with a random integer value. It then goes back to the beginning of the file and reads all of the records, deleting any record whose field value is under 100. Note that the calls to *rf.insert* will allocate as many new blocks as necessary to hold the records; the client code, however, has no idea that this is happening.

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();
Schema sch = new Schema();
sch.addIntField("A");
TableInfo ti = new TableInfo("junk", sch);
RecordFile rf = new RecordFile(ti, tx);
for (int i=0; i<10000; i++) {
      rf.insert();
      int n = (int) Math.round(Math.random() * 200);
      rf.setInt("A", n);
}

int count = 0;
rf.beforeFirst();
while (rf.next()) {
      if (rf.getInt("A") < 100) {
            count++;
            rf.delete();
      }
}
System.out.println(count + " values under 100 were deleted");
rf.close();
tx.commit();
```

Figure 15-20: Inserting many records into a file

15.3.7 Implementing record files

The code for class *RecordFile* appears in Figure 15-21.

```
public class RecordFile {
   private TableInfo ti;
   private Transaction tx;
   private String filename;
   private RecordPage rp;
   private int currentblknum;

   public RecordFile(TableInfo ti, Transaction tx) {
      this.ri = ti;
      this.tx = tx;
      filename = ti.fileName();
      if (tx.size(filename) == 0)
         appendBlock();
      moveTo(0);
   }

   public void close() {
      rp.close();
   }

   public void beforeFirst() {
      moveTo(0);
   }

   public boolean next() {
      while (true) {
         if (rp.next())
            return true;
         if (atLastBlock())
            return false;
         moveTo(currentblknum + 1);
      }
   }

   public int getInt(String fldname) {
      return rp.getInt(fldname);
   }

   public String getString(String fldname) {
      return rp.getString(fldname);
   }

   public void setInt(String fldname, int val) {
      rp.setInt(fldname, val);
   }

   public void setString(String fldname, String val) {
      rp.setString(fldname, val);
   }

   public void delete() {
      rp.delete();
   }

   public void insert() {
      while (!rp.insert()) {
         if (atLastBlock())
```

```
            appendBlock();
         moveTo(currentblknum + 1);
      }
   }

   public void moveToRid(RID rid) {
      moveTo(rid.blockNumber());
      rp.moveToId(rid.id());
   }

   public RID currentRid() {
      int id = rp.currentId();
      return new RID(currentblknum, id);
   }

   private void moveTo(int b) {
      if (rp != null)
         rp.close();
      currentblknum = b;
      Block blk = new Block(filename, currentblknum);
      rp = new RecordPage(blk, ti, tx);
   }

   private boolean atLastBlock() {
      return currentblknum == tx.size(filename) - 1;
   }

   private void appendBlock() {
      RecordFormatter fmtr = new RecordFormatter(ti);
      tx.append(filename, fmtr);
   }
}
```

Figure 15-21: The code for the SimpleDB class *RecordFile*

A *RecordFile* object holds the record page for its current block. The *get*/*set*/*delete* methods simply call the corresponding method of the record page. The private method *moveTo* is called when the current block changes; that method closes the current record page, and opens another one for the specified block.

The algorithm for the *next* method is:

- Move to the next record in the current record page.
- If there are no more records in that page, then move to the next block of the file and get its next record.
- Continue until either a next record is found, or the end of the file is encountered.

It is possible for multiple blocks of a file to be empty (see Exercise 15.2), so it may be necessary to loop through several blocks.

The *insert* method tries to insert a new record beginning at the current block, moving through the file until a non-full block is found. If all blocks are full, then it appends a new block to the file and inserts the record there.

RID objects are simply a combination of two integers: a block number and a record id. The code for the class *RID* is therefore straightforward, and appears in Figure 15-22.

```
public class RID {
   private int blknum;
   private int id;

   public RID(int blknum, int id) {
      this.blknum = blknum;
      this.id     = id;
   }

   public int blockNumber() {
      return blknum;
   }

   public int id() {
      return id;
   }

   public boolean equals(RID r) {
      return blknum == r.blknum && id==r.id;
   }
}
```

Figure 15-22: The code for the SimpleDB class *RID*

15.4 Chapter Summary

- The record manager is the portion of the database system that stores records in a file. It has three basic responsibilities:
  - Placing fields within records.
  - Placing records within blocks.
  - Providing access to the records in a file.

There are several issues that must be addressed when designing a record manager.

- One issue is whether to support *variable-length fields*. Fixed-length records can be implemented easily, because fields can be updated in place. Updating a variable-length field can cause records to spill out of a block and be placed into an *overflow block*.

- SQL has three different string types: *char, varchar*, and *clob*.
  - The *char* type is most naturally implemented using a fixed-length representation;
  - the *varchar* type is most naturally implemented using a variable-length representation;
  - and the *clob* type is implemented most naturally using a fixed-length representation that stores the string in an auxiliary file.

- A common implementation technique for variable-length records is to use an *ID-table*. Each entry in the table points to a record in the page. A record can move around in a page by just changing its entry in the ID-table.

- A second issue is whether to create *spanned records*. Spanned records are useful because they do not waste space and allow for large records, but they are more complicated to implement.

- A third issue is whether to allow *non-homogeneous records* in a file. Non-homogeneous records allow related records to be *clustered* on a page. Clustering can lead to very efficient joins, but tend to make other queries more expensive. The record manager can implement non-homogeneous records by storing a *tag field* at the beginning of each record; the tag denotes the table that the record belongs to.

- A fourth issue is how to determine the offset of each field within a record. The record manager may need to pad the fields so that they are *aligned* on appropriate byte boundaries. A field in a fixed-length record has the same offset for each record. It may be necessary to search a variable-length record for the beginning of its fields.

15.5 Suggested Reading

The ideas and techniques in this chapter have been present in relational databases from the very beginning. Section 3.3 of [Stonebraker et al. 1976] describes the approach taken by the first version of INGRES; this approach uses the variation of the ID table described in Section 15.2.2. Section 3 of [Astrahan et al. 1976] describes the page structure for the early System R database system (which later became IBM’s DB2 product), which stored records non-homogeneously. Both articles discuss a broad range of implementation ideas, and are well worth reading in their entirety. A more detailed discussion of these techniques, together with a C-based implementation of an example record manager, appears in Chapter 14 of [Gray and Reuter 1993].

The strategy of storing each record contiguously in a page is not necessarily best. The article [Ailamaki et al. 2002] advocates breaking up the records on a page, placing the values for each field together. Although this record organization doesn’t change the number of disk accesses performed by the record manager, it significantly improves the performance of the CPU, by utilizing its data cache more effectively. The article [Stonebraker et al. 2005] goes even farther, by proposing that all records in the table should be split up so that the values for each field are stored together. The article shows how column-based storage can be more compact than record-based storage, which can lead to more efficient queries.

An implementation strategy for very large records is described in [Carey et al. 1986].

15.6 Exercises

CONCEPTUAL PROBLEMS

15.1 Assume that the block size is 400 bytes and that records cannot span blocks. Calculate the maximum number of records that can fit in a SimpleDB record page and the amount of wasted space in the page for each of the following record sizes: 10 bytes, 20 bytes, 50 bytes, 100 bytes. (Don't forget about the 4-byte INUSE/EMPTY flag.)

15.2 Explain how a record file can contain blocks having no records.

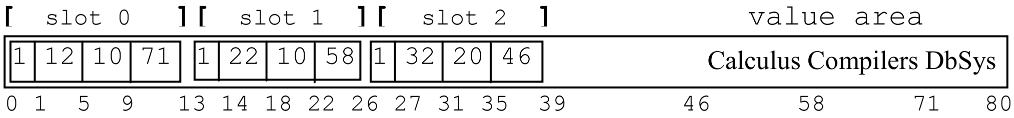
15.3 Consider each table in the university database (except STUDENT).
a) Give the table information for that table. (You can use the *varchar* declarations in the demo client files, or assume that all string fields are defined as *varchar(20)*.)
b) Draw a picture of the record page(s) (as in Figure 15-5) for each table, using the records of Figure 1-1. As in Figure 15-5, assume that the empty/full flag is a single byte long. Also assume a fixed-length implementation of string fields.
c) Do part (b), but assume a variable-length implementation of string fields. Use Figure 15-8(c) as a model.
d) Revise your pictures from parts (b) and (c) to show the state of the pages after their second record has been deleted.

15.4 Another way to deal with very large strings is to not store them in the database. Instead, we could place the strings in an OS file, and store the name of the file in the database. This strategy would eliminate the need for the *clob* type. Give several reasons why this strategy is not particularly good.

15.5 Suppose that you want to insert a record into a block that contains an overflow block, as in Figure 15-7(b). Is it a good idea to save the record in the overflow block? Explain.

15.6 Consider the code fragment of Figure 15-15. Suppose that the call to *tx.commit()* was moved to immediately after the constructor for *RecordPage*. What problems could occur?

15.7 Here is another way to implement variable-length records. Each block has two areas: a sequence of fixed-length slots (as in SimpleDB), and a place where variable-length values are stored. A record is stored in a slot. Its fixed-length values are stored with the record, and its variable-length values are stored in the value area. The record will contain the block offset where the value is located. For example, the records in Figure 15-8(a) could be stored like this:



a) Explain what should happen when a variable-length value gets modified. Do you need an overflow block? What should it look like?
b) Compare this storage strategy with that of ID tables. Explain the comparative benefits of each.

c) Which implementation strategy do you prefer? Why?

15.8 Using a byte for each empty/inuse flag wastes space, since only a bit is needed. An alternative implementation strategy is to store the empty/inuse bits for each slot together in a bit array at the beginning of the block. This bit array could be implemented as one or more 4-byte integers.
a) Compare this bit array with the ID table of Figure 15-8(c).
b) Suppose that the block size is 4K and records are assumed to be at least 15 bytes. How many integers are needed to store the bit array?
c) Describe an algorithm for finding an empty slot to insert a new record.
d) Describe an algorithm for finding the next non-empty record in a block.

15.9 Consider the following scenario:

1. Transaction T_1 deletes a record from a block. The record manager sets the record's flag to EMPTY.
2. Transaction T_2 inserts a record into the block. The record manager chooses the deleted record, setting its flag to INUSE.
3. Transaction T_2 assigns values to fields of the new record.
4. Transaction T_1 does a rollback.

Since the old values of the deleted record were not saved in the log, the record cannot be restored to its original value. This is a serious problem.
a) Explain why this scenario cannot happen in SimpleDB.
b) If transactions T_1 and T_2 attempt to perform the above steps, what actually happens in SimpleDB?

PROGRAMMING PROBLEMS

15.10 SimpleDB only knows how to read files in the forward direction.
a) Revise the classes *RecordFile* and *RecordPage* to support a *previous* method, as well as the method *afterLast*, which positions the current record to be after the last record in the file (or page).
b) Write a test program to print the records of the STUDENT file in reverse order.

15.11 Revise the class *RecordPage* so that its block is not pinned by the constructor, but instead is pinned at the beginning of each *get/set* method. Similarly, the block is unpinned at the end of each *get/set* method, thereby eliminating the need for a *close* method. Do you think this is better than the SimpleDB implementation? Explain.

15.12 Revise the record manager so that *varchar* fields have a variable-length implementation.

15.13 Revise the record manager so that records are spanned.

15.14 Revise the class *TableInfo* to pad string fields so that their size is always a multiple of 4.

15.15 Write programs that test the *RecordPage* and *RecordFile* classes. For example, implement the example code fragments in the chapter.

15.16 Revise the SimpleDB record manager to handle null field values. Since it is unreasonable to use a particular integer or string value to denote a null, we shall use flags to specify which values are null. In particular, suppose that a record contains N fields. Then we shall store N additional bits with each record, such that the value of the ith bit is 1 iff the value of the ith field is null. If we assume that N<32, then we can use the EMPTY/INUSE integer for this purpose. Bit 0 of this integer denotes empty/inuse, as before. But now we use the other bits to hold null-value information. You should make the following revisions to the code:

- Modify *TableInfo* so that it has a method *bitLocation(fldname)*, which returns the position in the flag where the field's null information bit is located.
- Modify *RecordPage* and RecordFile to have two additional public methods: a void method *setNull(fldname)*, which stores a 1 in the appropriate bit of the flag; and a boolean method *isNull(fldname)*, which returns true if the null-bit for the specified field of the current record is 1.
- Modify the *insert* method of *RecordPage* so that all of the fields of the new record are initially null.

16

METADATA MANAGEMENT

and the package simpledb.metadata

The previous chapter examined how the record manager stores records in files. As we saw, however, a file is useless by itself; the record manager also needs to know the records' table information in order to "decode" the contents of each block. This table information is called *metadata*. In this chapter we will examine the kinds of metadata supported by a database system, their purpose and functionality, and the ways in which the system stores metadata in the database.

16.1 The Metadata Manager

> *Metadata is the information about a database, apart from its contents.*

Database systems maintain a wide variety of metadata. For example:

- *Table* metadata describes the structure of the table's records, such as the length, type, and offset of each field. The table information used by the record manager is an example of this kind of metadata.
- *View* metadata describes the properties of each view, such as its definition and creator. This metadata helps the planner process queries that mention views.
- *Index* metadata describes the indexes that have been defined on the table. The planner uses this metadata to see if a query can be evaluated using an index.
- *Statistical* metadata describes the size of each table and the distribution of its field values. This metadata is important for estimating the cost of a query.

The metadata for the first three categories is generated when a table, view, or index is created. Statistical metadata is generated each time the database is updated.

> *The metadata manager is the portion of the database system*
> *that stores and retrieves its metadata.*

The metadata manager saves its metadata in the *system catalog*. It also provides methods so that clients can retrieve the metadata from the catalog. The SimpleDB metadata manager is implemented by the class *MetadataMgr*, whose API appears in Figure 16-1.

MetadataMgr

```
public void createTable(String tblname, Schema sch,
                        Transaction tx);
public TableInfo getTableInfo(String tblname, Transaction tx);

public void createView(String viewname, String viewdef,
                       Transaction tx);
public String getViewDef(String viewname, Transaction tx);

public void createIndex(String idxname, String tblname,
                        String fldname, Transaction tx);
public Map<String,IndexInfo> getIndexinfo(String tblname,
                                          Transaction tx);

public StatInfo getStatInfo(String tblname, TableInfo ti,
                            Transaction tx);
```

Figure 16-1: The API for the SimpleDB metadata manager

This API contains two methods for each type of metadata – One method generates and saves the metadata, and the other method retrieves it. The only exception is for statistical metadata, whose generation method is called internally, and is thus private.

SimpleDB has one *MetadataMgr* object for the entire system; this object is available from the static method *mdMgr* in the class *SimpleDB*.

The remaining sections of this chapter will cover the management of table, view, statistical, and index metadata, using SimpleDB as an example.

16.2 Table Metadata

16.2.1 Using table metadata

To create a table, a client calls the method *createTable*, passing in the table's name and schema; the method calculates additional metadata for the table (such as the record offsets and the name of the file where the table is stored), and saves it in the catalog. The method *getTableInfo* goes to the catalog, extracts the metadata for the specified table, puts it into a *TableInfo* object, and returns the object.

The code fragment of Figure 16-2 demonstrates these methods. Part 1 of the code creates the table DEPT as it was defined in Figure 1-1. This table has two fields: an integer field named *DId*, and a *varchar(8)* string field named *DName*. Part 2 of this code (which typically would occur in a different program) retrieves the table's metadata from the catalog as a *TableInfo* object. It then opens a record file for the DEPT table and prints the name of each department.

```
SimpleDB.init("studentdb");
MetadataMgr mdMgr = SimpleDB.mdMgr();

// Part 1: Create the DEPT table
Transaction tx1 = new Transaction();
Schema sch = new Schema();
sch.addIntField("did");
sch.addStringField("dname", 8);
mdMgr.createTable("dept", sch, tx1);
tx1.commit();

// Part 2:  Print the name of each department
Transaction tx2 = new Transaction();
TableInfo ti = mdMgr.getTableInfo("dept", tx2);
RecordFile rf = new RecordFile(ti, tx2);
while (rf.next())
      System.out.println(rf.getString("dname"));
rf.close();
tx2.commit();
```

Figure 16-2: Using the table metadata methods

16.2.2 Implementing table metadata

The metadata manager saves metadata in the database's catalog. But how does it implement the catalog? The approach typically taken by database systems is to use database tables to store the catalog.

A database system stores its metadata in catalog tables.

SimpleDB holds its table metadata in two catalog tables: the table *tblcat* stores metadata that is specific to each table, and the table *fldcat* stores metadata that is specific to each field of each table. These tables have the following schemas (with key fields underlined):

```
tblcat(TblName, RecLength)
fldcat(TblName, FldName, Type, Length, Offset)
```

There is one record in the *tblcat* table for each database table, and one record in the *fldcat* table for each field of each table. The *RecLength* field gives the length of the record in bytes, as calculated by *TableInfo*. The *Length* field gives the length of the field in characters, as specified in the schema. For an example, the catalog tables corresponding to the database of Figure 1-1 are shown in Figure 16-3[†]. Note how the table's schema information has been "flattened" into a series of *fldcat* records. The field *Type* in table *fldcat* contains the values 4 and 12; these values are the codes for types INTEGER and VARCHAR, as defined in the JDBC class *Types*.

[†] The offsets in Figure 16-3 are illustrative. The actual offsets generated by SimpleDB depend on the order in which the schema stores its fields. Since the current implementation uses a hash map, the fields will be stored in a somewhat random order.

tblcat	TblName	RecLength
	student	26
	dept	16
	course	32
	section	28
	enroll	18

fldcat	TblName	FldName	Type	Length	Offset
	student	sid	4	0	0
	student	sname	12	10	4
	student	majorid	4	0	18
	student	gradyear	4	0	22
	dept	did	4	0	0
	dept	dname	12	8	4
	course	cid	4	0	0
	course	title	12	20	4
	course	deptid	4	0	28
	section	sectid	4	0	0
	section	courseid	4	0	4
	section	prof	12	8	8
	section	year	4	0	20
	enroll	eid	4	0	0
	enroll	studentid	4	0	4
	enroll	sectionid	4	0	8
	enroll	grade	12	2	12

Figure 16-3: Catalog tables for the university database

The catalog tables are actual, honest-to-goodness database tables, and thus can be accessed the same as any user-created table. For example, the SQL query of Figure 16-4 retrieves the names and length of all fields in the STUDENT table[†]:

[†] Note that the constant ‘student’ is in lower case, even though the table was defined in upper case. The reason is that all table and field names in SimpleDB are stored in lower case, and constants in SQL statements are case-sensitive.

```
      select FldName, Length
      from fldcat
      where TblName = 'student'
```

Figure 16-4: Using an SQL query to retrieve metadata

The catalog tables also contain records that encode their own metadata. These records do not appear in Figure 16-3; instead, Exercise 16.1 asks you to determine them.

SimpleDB implements the table-metadata methods in the class *TableMgr*, whose code appears in Figure 16-5.

```
public class TableMgr {
   // table and field names are varchar(16)
   public static final int MAX_NAME = 16;

   private TableInfo tcatInfo, fcatInfo;

   public TableMgr(boolean isNew, Transaction tx) {
      Schema tcatSchema = new Schema();
      tcatSchema.addStringField("tblname", MAX_NAME);
      tcatSchema.addIntField("reclength");
      tcatInfo = new TableInfo("tblcat", tcatSchema);

      Schema fcatSchema = new Schema();
      fcatSchema.addStringField("tblname", MAX_NAME);
      fcatSchema.addStringField("fldname", MAX_NAME);
      fcatSchema.addIntField("type");
      fcatSchema.addIntField("length");
      fcatSchema.addIntField("offset");
      fcatInfo = new TableInfo("fldcat", fcatSchema);

      if (isNew) {
         createTable("tblcat", tcatSchema, tx);
         createTable("fldcat", fcatSchema, tx);
      }
   }

   public void createTable(String tblname, Schema sch, Transaction tx){
      TableInfo ti = new TableInfo(tblname, sch);

      // insert one record into tblcat
      RecordFile tcatfile = new RecordFile(tcatInfo, tx);
      tcatfile.insert();
      tcatfile.setString("tblname", tblname);
      tcatfile.setInt("reclength", ti.recordLength());
      tcatfile.close();

      // insert a record into fldcat for each field
      RecordFile fcatfile = new RecordFile(fcatInfo, tx);
      for (String fldname : sch.fields()) {
         fcatfile.insert();
         fcatfile.setString("tblname", tblname);
         fcatfile.setString("fldname", fldname);
         fcatfile.setInt   ("type", sch.type(fldname));
         fcatfile.setInt   ("length", sch.length(fldname));
         fcatfile.setInt   ("offset", ti.offset(fldname));
      }
      fcatfile.close();
   }

   public TableInfo getTableInfo(String tblname, Transaction tx) {
      RecordFile tcatfile = new RecordFile(tcatInfo, tx);
      int reclen = -1;
      while (tcatfile.next())
         if(tcatfile.getString("tblname").equals(tblname)) {
            reclen = tcatfile.getInt("reclength");
            break;
         }
      tcatfile.close();

      RecordFile fcatfile = new RecordFile(fcatInfo, tx);
      Schema sch = new Schema();
      Map<String,Integer> offsets = new HashMap<String,Integer>();
      while (fcatfile.next())
         if (fcatfile.getString("tblname").equals(tblname)) {
            String fldname = fcatfile.getString("fldname");
            int fldtype    = fcatfile.getInt("type");
            int fldlen     = fcatfile.getInt("length");
            int offset     = fcatfile.getInt("offset");
            offsets.put(fldname, offset);
            sch.addField(fldname, fldtype, fldlen);
         }
      fcatfile.close();
      return new TableInfo(tblname, sch, offsets, reclen);
   }
}
```

Figure 16-5: The code for the SimpleDB class *TableMgr*

Its constructor is called during system startup. The constructor creates the schemas for *tblcat* and *fldcat*, and calculates their *TableInfo* objects. If the database is new, it also creates the two catalog tables.

The *createTable* method uses a record file to insert records into each catalog table. It inserts one record into *tblcat* corresponding to the table, and one record into *fldcat* for each field of the table.

The *getTableInfo* method opens a record file on the two catalog tables, and scans it for records corresponding to the specified table name. Those records are then used to construct the *TableInfo* object for that table.

16.3 View Metadata

A *view* is a table whose records are computed dynamically from a query. That query is called the *definition* of the view, and is specified when the view is created. The metadata manager stores the definition of each newly-created view, and retrieves its definition when requested.

The SimpleDB class *ViewMgr* handles this responsibility; its code appears in Figure 16-6.

```
class ViewMgr {
   private static final int MAX_VIEWDEF = 100;
   TableMgr tblMgr;

   public ViewMgr(boolean isNew, TableMgr tblMgr, Transaction tx) {
      this.tblMgr = tblMgr;
      if (isNew) {
         Schema sch = new Schema();
         sch.addStringField("viewname", TableMgr.MAX_NAME);
         sch.addStringField("viewdef", MAX_VIEWDEF);
         tblMgr.createTable("viewcat", sch, tx);
      }
   }

   public void createView(String vname, String vdef, Transaction tx) {
      TableInfo ti = tblMgr.getTableInfo("viewcat", tx);
      RecordFile rf = new RecordFile(ti, tx);
      rf.insert();
      rf.setString("viewname", vname);
      rf.setString("viewdef", vdef);
      rf.close();
   }

   public String getViewDef(String vname, Transaction tx) {
      String result = null;
      TableInfo ti = tblMgr.getTableInfo("viewcat", tx);
      RecordFile rf = new RecordFile(ti, tx);
      while (rf.next())
         if (rf.getString("viewname").equals(vname)) {
            result = rf.getString("viewdef");
            break;
         }
      rf.close();
      return result;
   }
}
```

Figure 16-6: The code for the SimpleDB class *ViewMgr*

The class stores view definitions in the catalog table *viewcat*, one record per view. The table has the following schema (the key is underlined):

```
viewcat(ViewName, ViewDef)
```

The class constructor is called during system startup, and creates the catalog table if the database is new. The methods *createView* and *getViewDef* both use a record file to access the catalog table – Method *createView* inserts a record into the table, and method *getViewDef* iterates through the table looking for the record corresponding to the specified view name. Note that view definitions are stored as *varchar* strings, which means that there is a relatively small limit on the length of a view definition. The current limit of 100 characters is, of course, completely unrealistic, as a view definition could be thousands of characters long. A better choice would be to implement the *ViewDef* field as a *clob* type, such as *clob(9999)*.

16.4 Statistical Metadata

16.4.1 The *StatInfo* API

Another form of metadata managed by a database system is the statistical information about each table in the database, such as how many records it has and the distribution of their field values. These statistics are used by the query planner to estimate costs. Experience has shown that a good set of statistics can significantly improve the execution time of queries; consequently, commercial metadata managers tend to maintain detailed, comprehensive statistics, such as value and range histograms for each field in each table, and correlation information between fields in different tables.

For simplicity, we shall consider only the following three kinds of statistical information:

- the number of blocks used by each table T;
- the number of records in each table T;
- for each field F of table T, the number of distinct F-values in T.

We denote these statistics by *B(T)*, *R(T)*, and *V(T,F)* respectively.

Figure 16-7 gives some example statistics for the student database. The values correspond to a university that admits about 900 students per year and offers about 500 sections per year; the university has kept this information for the last 50 years.

T	B(T)	R(T)	V(T,F)
STUDENT	4,500	45,000	45,000 for F=SId 44,960 for F=SName 50 for F=GradYear 40 for F=MajorId
DEPT	2	40	40 for F=DId, DName
COURSE	25	500	500 for F=CId, Title 40 for F=DeptId
SECTION	2,500	25,000	25,000 for F=SectId 500 for F=CourseId 250 for F=Prof 50 for F=YearOffered
ENROLL	50,000	1,500,000	1,500,000 for F=EId 25,000 for F=SectionId 45,000 for F=StudentId 14 for F=Grade

Figure 16-7: Example statistics about the university database

The values in Figure 16-7 try to be realistic, but do not necessarily correspond to values that might be calculated from Figure 1-1. Instead, the figures assume that 10 STUDENT records fit per block, 20 DEPT records per block, and so on.

Look at the V(T,F) values for the STUDENT table. The fact that *SId* is a key of STUDENT means that V(STUDENT, *SId*) = 45,000. The assignment V(STUDENT, *SName*) = 44,960 means that 40 of the 45,000 students have had duplicate names. The assignment V(STUDENT, *GradYear*) = 50 means that at least one student graduated in each of the last 50 years. And the assignment V(STUDENT, *MajorId*) = 40 means that each of the 40 departments has had at least one major at some point.

SimpleDB implements the statistical functions B(T), R(T), and V(T,F) by the methods *blocksAccessed*, *recordsOutput*, and *distinctValues* in the class *StatInfo*. Its API appears in Figure 16-8.

StatInfo

```
public int blocksAccessed();
public int recordsOutput();
public int distinctValues(String fldname);
```

Figure 16-8: The API for SimpleDB table statistics

The metadata manager method *getStatInfo* returns a *StatInfo* object for the specified table. For example, consider the code fragment in Figure 16-9. This code obtains the statistics for the STUDENT table and prints the value of B(STUDENT), R(STUDENT), and V(STUDENT, MajorId).

```
SimpleDB.init("studentdb");
MetadataMgr mdMgr = SimpleDB.mdMgr();

Transaction tx = new Transaction();
TableInfo ti   = mdMgr.getTableInfo("student", tx);
StatInfo si    = mdMgr.getStatInfo(ti, tx);
System.out.println(si.blocksAccessed() + " " +
                   si.recordOutput()   + " " +
                   si.distinctValues("majorid"));
tx.commit();
```

Figure 16-9: Obtaining and printing statistics about a table

16.4.2 Implementing statistical metadata

A database system can manage statistical metadata in one of two ways. One way is to store the information in the database catalog, updating it whenever the database changes. The other is to store the information in memory, calculating it when the server is initialized.

The first approach corresponds to creating two new catalog tables, called *tblstats* and *fldstats*, having the following schemas.

```
tblstats(TblName, NumBlocks, NumRecords)
fldstats(TblName, FldName, NumValues)
```

The *tblstats* table would have one record for each table T, which would contain the values B(T) and R(T). The *fldstats* table would have one record for each field F of each table T, containing the value V(T,F). The problem with this approach is the cost of keeping the statistics up-to-date. Every call to *insert*, *delete*, *setInt*, and *setString* would have to update these tables. Additional disk accesses would be required to write the modified pages to disk and write the corresponding log records. Moreover, concurrency would be reduced – every update to table T would xlock the blocks containing T's statistical records, which would force the transactions that need to read T's statistics (as well as the statistics of the other tables having records on the same pages) to wait.

One viable solution to this problem is to let transactions read the statistics without obtaining slocks, as in the read-uncommitted isolation level of Section 14.4.7. The loss of accuracy is tolerable, because the database system uses these statistics to compare the estimated execution times of query plans. The statistics therefore do not need to be accurate as long as the estimates they produce are reasonable.

The second implementation strategy is to forget about catalog tables, and to store the statistics directly in memory. The statistical data is relatively small, and should fit easily in main memory. The only problem is that the statistics will need to be computed from scratch each time the server starts. This calculation requires a scan of each table in the database, counting the number of records, blocks, and values seen. If the database is not too large, this computation will not delay the system startup too much.

This main-memory strategy has two options for dealing with database updates. The first option is for each update to the database to update the statistics, as before. The second option is to leave the statistics un-updated, but to recalculate them, from scratch, every so often. This second option relies again on the fact that accurate statistical information is not necessary, and so it is tolerable to let the statistics get a bit out of date before refreshing them.

SimpleDB adopts the second option of the second approach. The class *StatMgr* keeps a variable (called *tableStats*) that holds cost information for each table. The class has a public method *statInfo* that returns the cost values for a specified table, and private methods *refreshStatistics* and *refreshTableStats* that recalculate the cost values. The code for the class appears in Figure 16-10.

```
class StatMgr {
   private Map<String,StatInfo> tablestats;
   private int numcalls;

   public StatMgr(Transaction tx) {
      refreshStatistics(tx);
   }

   public synchronized StatInfo getStatInfo(String tblname,
                                        TableInfo ti, Transaction tx) {
      numcalls++;
      if (numcalls > 100)
         refreshStatistics(tx);
      StatInfo si = tablestats.get(tblname);
      if (si == null) {
         refreshTableStats(tblname, tx);
         si = tablestats.get(tblname);
      }
      return si;
   }

   private synchronized void refreshStatistics(Transaction tx) {
      tablestats = new HashMap<String,StatInfo>();
      numcalls = 0;
      TableInfo tcatinfo = SimpleDB.mdMgr().getTableInfo("tblcat", tx);
      RecordFile tcatfile = new RecordFile(tcatinfo, tx);
      while(tcatfile.next()) {
         String tblname = tcatfile.getString("tblname");
         refreshTableStats(tblname, tx);
      }
      tcatfile.close();
   }

   private synchronized void refreshTableStats(String tblname,
                                               Transaction tx) {
      int numRecs = 0;
      TableInfo ti = SimpleDB.mdMgr().getTableInfo(tblname, tx);
      RecordFile rf = new RecordFile(ti, tx);
      while (rf.next())
         numRecs++;
      int blknum = rf.currentRid().blockNumber();
      rf.close();
      int numblocks = 1 + blknum;
      StatInfo si = new StatInfo(numblocks, numRecs);
      tablestats.put(tblname, si);
   }
}
```

Figure 16-10: The code for the SimpleDB class *StatMgr*

The class *StatMgr* keeps a counter that is incremented each time *statInfo* is called. If the counter reaches a particular value (here, 100), then *refreshStatistics* is called to recalculate the cost values for all tables. If *statInfo* is called on a table for which there are no known values, then *refreshTableStats* is called to calculate the statistics for that table.

The code for *refreshStatistics* loops through the *tblcat* table. The body of the loop extracts the name of a table and calls *refreshTableStats* to calculate the statistics for that table. The *refreshTableStats* method loops through the contents of that table, counting records, and calls *size* to determine the number of blocks used. For simplicity, the method does not count field values. Instead, the *StatInfo* object makes a wild guess at the number of distinct values for a field, based on the number of records in its table.

The class *StatInfo* holds the cost information for a table; its code appears in Figure 16-11. Note that *distinctValues* does not use the field value passed into it, because it naïvely assumes that approximately 1/3 of the values of any field are distinct. Needless to say, this assumption is pretty bad. Exercise 16.12 asks you to rectify the situation.

```
public class StatInfo {
   private int numBlocks;
   private int numRecs;

   public StatInfo(int numblocks, int numrecs) {
      this.numBlocks = numblocks;
      this.numRecs   = numrecs;
   }

   public int blocksAccessed() {
      return numBlocks;
   }

   public int recordsOutput() {
      return numRecs;
   }

   public int distinctValues(String fldname) {
      return 1 + (numRecs / 3);
   }
}
```

Figure 16-11: The code for the SimpleDB class *StatInfo*

16.5 Index Metadata

16.5.1 Using index metadata

The metadata for an index consists of its name, the name of the table it is indexing, and the list of its indexed fields. The *index manager* is the portion of the system that stores and retrieves this metadata. The index manager supports two methods. When an index is created, the *createIndex* method stores its metadata in the catalog, and when information about an index is requested, the *getIndexInfo* method retrieves its metadata from the catalog.

In SimpleDB, the method *getIndexInfo* returns a map of *Indexinfo* objects, keyed by the indexed field. An *IndexInfo* object holds the metadata for a single index. Its API appears in Figure 16-12.

<u>IndexInfo</u>

```
   public IndexInfo(String idxname, String tblname, String fldname,
                    Transaction tx);
   public int blocksAccessed();
   public int recordsOutput();
   public int distinctValues(String fldname);
   public Index open();
```

Figure 16-12: The API for SimpleDB index metadata

The constructor takes in the metadata for the index. SimpleDB indexes can have only one indexed field, which is why the third argument is a string (and not a list of strings). The methods *blocksAccessed*, *recordsOutput*, and *distinctValues* provide statistical information about the index, similar to the class *StatInfo*. The method *blocksAccessed* returns the number of block accesses required to search the index (not the size of the index). Methods *recordsOutput* and *distinctValues* return the number of records in the index and the number of distinct values of the indexed field, which are the same values as in the indexed table.

An *IndexInfo* object also has the method *open*. This method uses the metadata to open the index, returning an *Index* object. The class *Index* contains methods to search the index, and is discussed in Chapter 21.

The code fragment of Figure 16-13 illustrates how the index manager is used. The code obtains the metadata for each index in the STUDENT table. It then loops through those indexes, printing their name and search cost. Finally, it opens the index on *MajorId*.

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();

MetadataMgr mdMgr = SimpleDB.mdMgr();
Map<String,IndexInfo> indexes = mdMgr.getIndexInfo("student", tx);

// Part 1: Print the name and cost of each index on STUDENT
for (String fldname : indexes.keySet()) {
   IndexInfo ii = indexes.get(fldname);
   System.out.println(fldname + "\t" + ii.blocksAccessed(fldname));
}

// Part 2:  Open the index on MajorId
IndexInfo ii = indexes.get("majorid");
Index idx = ii.open();
```

Figure 16-13: Using the SimpleDB index manager

16.5.2 Implementing index metadata

The SimpleDB class *IndexMgr* implements the index manager; see Figure 16-14. SimpleDB stores index metadata in the catalog table *idxcat*. This table has one record for each index, and three fields: the name of the index, the name of the table being indexed, and the name of the indexed field.

```
public class IndexMgr {
   private TableInfo ti;

   public IndexMgr(boolean isnew, TableMgr tblmgr, Transaction tx) {
      if (isnew) {
         Schema sch = new Schema();
         sch.addStringField("indexname", MAX_NAME);
         sch.addStringField("tablename", MAX_NAME);
         sch.addStringField("fieldname", MAX_NAME);
         tblmgr.createTable("idxcat", sch, tx);
      }
      ti = tblmgr.getTableInfo("idxcat", tx);
   }

   public void createIndex(String idxname, String tblname,
                           String fldname, Transaction tx) {
      RecordFile rf = new RecordFile(ti, tx);
      rf.insert();
      rf.setString("indexname", idxname);
      rf.setString("tablename", tblname);
      rf.setString("fieldname", fldname);
      rf.close();
   }

   public Map<String,IndexInfo> getIndexInfo(String tblname,
                                             Transaction tx) {
      Map<String,IndexInfo> result = new HashMap<String,IndexInfo>();
      RecordFile rf = new RecordFile(ti, tx);
      while (rf.next())
         if (rf.getString("tablename").equals(tblname)) {
            String idxname = rf.getString("indexname");
            String fldname = rf.getString("fieldname");
            IndexInfo ii = new IndexInfo(idxname, tblname, fldname,
                                         tx);
            result.put(fldname, ii);
         }
      rf.close();
      return result;
   }
 }
```

Figure 16-14: The code for the SimpleDB class *IndexMgr*

The constructor is called during system startup, and creates the catalog table if the database is new. The code for methods *createIndex* and *getIndexInfo* is straightforward. Both methods open a record file on the catalog table. The method *createIndex* inserts a new record into the record file. The method *getIndexInfo* searches the record file for those records having the specified table name.

The code for the class *IndexInfo* appears in Figure 16-15.

```
public class IndexInfo {
   private String idxname, fldname;
   private int idxtype;
   private Transaction tx;
   private TableInfo ti;
   private StatInfo si;

   public IndexInfo(String idxname, String tblname, String fldname,
                    Transaction tx) {
      this.idxname = idxname;
      this.fldname = fldname;
      this.tx = tx;
      ti = SimpleDB.mdMgr().getTableInfo(tblname, tx);
      si = SimpleDB.mdMgr().getStatInfo(tblname, ti, tx);
   }

   public Index open() {
      Schema sch = schema();
      return new BTreeIndex(idxname, sch, tx);
   }

   public int blocksAccessed() {
      int rpb = BLOCK_SIZE / ti.recordLength();
      int numblocks = si.recordsOutput() / rpb;
      return BTreeIndex.searchCost(numblocks, rpb);
   }

   public int recordsOutput() {
      return si.recordsOutput() / si.distinctValues(fldname);
   }

   public int distinctValues(String fname) {
      if (fldname.equals(fname))
         return 1;
      else
         return Math.min(si.distinctValues(fldname), recordsOutput());
   }

   private Schema schema() {
      Schema sch = new Schema();
      sch.addIntField("block");
      sch.addIntField("id");
      if (ti.schema().type(fldname) == INTEGER)
         sch.addIntField("dataval");
      else {
         int fldlen = ti.schema().length(fldname);
         sch.addStringField("dataval", fldlen);
      }
      return sch;
   }
}
```

Figure 16-15: The code for the SimpleDB class *IndexInfo*

The *IndexInfo* constructor provides it with the name of the index, the table being indexed, and the indexed field. In addition to this explicit metadata, an *IndexInfo* object also has variables holding the structural and statistical metadata of its associated table; this metadata allows the *IndexInfo* object to construct the schema for the index record, and to estimate the size of the index file.

The method *open* opens the index, by passing the index name and schema to the *BTreeIndex* constructor. The class *BTreeIndex* implements the index, and is discussed in Chapter 21. The method *blocksAccessed* estimates the search cost of the index. It first uses the index's *TableInfo* information to determine the length of each index record and estimate the records per block (RPB) of the index and the size of the index file. Then it calls the index-specific method *searchCost* to calculate the number of block accesses for that index type. The method *recordsOutput* estimates the number of index records matching a search key. And the method *distinctValues* returns the same value as in the indexed table.

16.6 Implementing the Metadata Manager

Metadata management in SimpleDB is implemented via the four separate manager classes *TableMgr*, *ViewMgr*, *StatMgr*, and *IndexMgr*. The class *MetadataMgr* hides this distinction, so that clients have a single place to obtain metadata; its code appears in Figure 16-16. This class is known as a *façade class*. Its constructor creates the four manager objects and saves them in private variables. Its methods replicate the public methods of the individual managers. When a client calls a method on the metadata manager, that method calls the appropriate local manager to do the work.

```
public class MetadataMgr {
   private static TableMgr  tblmgr;
   private static ViewMgr   viewmgr;
   private static IndexMgr  idxmgr;
   private static StatMgr   statmgr;

   public MetadataMgr(boolean isnew, Transaction tx) {
      tblmgr  = new TableMgr(isnew, tx);
      viewmgr = new ViewMgr(isnew, tblmgr, tx);
      idxmgr  = new IndexMgr(isnew, tblmgr, tx);
      statmgr = new StatMgr(tblmgr, tx);
   }

   public void createTable(String tblname, Schema sch, Transaction tx){
      tblmgr.createTable(tblname, sch, tx);
   }

   public TableInfo getTableInfo(String tblname, Transaction tx) {
      return tblmgr.getTableInfo(tblname, tx);
   }

   public void createView(String viewname, String viewdef,
                                           Transaction tx) {
      viewmgr.createView(viewname, viewdef, tx);
   }

   public String getViewDef(String viewname, Transaction tx) {
      return viewmgr.getViewDef(viewname, tx);
   }

   public void createIndex(String idxname, String tblname,
                           String fldname, Transaction tx) {
      idxmgr.createIndex(idxname, tblname, fldname, tx);
   }

   public Map<String,IndexInfo> getIndexInfo(String tblname,
                                             Transaction tx) {
      return idxmgr.getIndexInfo(tblname, tx);
   }

   public StatInfo getStatInfo(String tblname, TableInfo ti,
                                               Transaction tx) {
      return statmgr.getStatInfo(tblname, ti, tx);
   }
}
```

Figure 16-16: The code for the SimpleDB class *MetadataMgr*

16.7 Chapter Summary

- *Metadata* is the information about a database, apart from its contents.

- The *metadata manager* is the portion of the database system that stores and retrieves its metadata.

- Database metadata in SimpleDB falls into four categories:
  - *Table* metadata describes the structure of the table's records, such as the length, type, and offset of each field.
  - *View* metadata describes the properties of each view, such as its definition and creator.
  - *Index* metadata describes the indexes that have been defined on the table.
  - *Statistical* metadata describes the size of each table and the distribution of its field values.

- The metadata manager saves its metadata in the *system catalog*. The catalog is often implemented as tables in the database, called *catalog tables*. Catalog tables can be queried the same as any other table in the database.

- Table metadata can be stored in two catalog tables – One table stores table information (such as the record length), and the other table stores field information (such as the name, length, and type of each field).

- View metadata consists primarily of the view definition, and can be saved in its own catalog table. The view definition will be an arbitrarily-long string, so a variable-length representation is appropriate.

- Statistical metadata holds information about the size and value distribution of each table in the database. Commercial database systems tend to maintain detailed, comprehensive statistics, such as value and range histograms for each field in each table, and correlation information between fields in different tables.

- A basic set of statistics consists of three functions:
  - B(T) returns the number of blocks used by table T.
  - R(T) returns the number of records in table T.
  - V(T,F) returns the number of distinct F-values in T.

- Statistics can be stored in catalog tables, or they can be calculated from scratch each time the database restarts. The former option avoids the long startup time, but can slow down the execution of transactions.

- Index metadata holds information on the name of each index, the table it is indexed on, and the indexed fields.

16.8 Suggested Reading

The catalog tables used in SimpleDB are about as small as possible, and similar to those used in the early INGRES system [Stonebraker et al. 1976]. On the other side of the spectrum, Oracle currently has such an extensive catalog that a 60-page book has been written to describe it [Kreines 2003].

Accurate and detailed statistical metadata is critical for good query planning. The approach taken in this chapter is crude, and commercial systems use much more sophisticated techniques. The article [Gibbons et al 2002] describes the use of histograms, and shows how they can be maintained efficiently in the face of frequent updates. Histogram information can be determined in various ways, one of the more interesting being via wavelet techniques [Matias et al. 1998]. It is even possible to collect statistics on previously-run queries, which can then be used to plan related queries [Bruno and Chaudhuri 2004].

16.9 Exercises

CONCEPTUAL EXERCISES

16.1 Give the *tblcat* and *fldcat* records that SimpleDB creates for the *tblcat* and *fldcat* tables. (HINT: Examine the code for *TableMgr*.)

16.2 Suppose that the only thing transaction T_1 does is create table X, and the only thing transaction T_2 does is create table Y.
a) What possible concurrent schedules can these transactions have?
b) Could T_1 and T_2 ever deadlock? Explain.

16.3 Standard SQL also allows a client to add a new field to an existing table. Give a good algorithm to implement this functionality.

PROGRAMMING EXERCISES

16.4 Standard SQL allows a client to remove a field from an existing table. Suppose that this functionality is implemented in a method called *TableMgr.removeField*.
a) One way to implement this method is to simply modify the field's record in *fldcat* to have a blank fieldname. Write the code for this method.
b) In part (a), none of the table's records are changed. What happens to their deleted field values? Why can't they ever be accessed?
c) Another way to implement this method is to remove the field's record from *fldcat*, and modify all of the existing data records in the table. This is considerably more work than in (a). Is it ever worth it? Explain the tradeoffs.

16.5 In the SimpleDB catalog tables, the field *tblname* of *tblcat* is its key, and the field *tblname* of *fldcat* is the corresponding foreign key. Another way to implement these tables would be to use an artificial key (say, *tblId*) for *tblcat*, with a corresponding foreign key in *fldcat* (say, named *tableId*).
a) Implement this design in SimpleDB.
b) Is this design better than the original one? (Does it save space? Does it save block accesses?)

16.6 Suppose that SimpleDB crashes while the catalog tables for a new database are being created.

a) Describe what will occur when the database is recovered after system restart. What problem arises?
b) Revise the SimpleDB code to fix this problem.

16.7 Write SimpleDB clients to do each of the following tasks, by querying the *tblcat* and *fldcat* tables directly:
a) Print the names and fields of all tables in the database (e.g., in the form of "T(A, B)").
b) Reconstruct and print the text of the SQL *create table* statement used to create a particular table (e.g., in the form of "create table T (A integer, B varchar(7))").

16.8 What happens when the method *getTableInfo* is called with a non-existent table name? Revise the code so that *null* is returned instead.

16.9 What problem can occur when a client creates a table with the same name as a table already in the catalog? Revise the code to prevent this from happening.

16.10 Revise *TableMgr* to have the method *dropTable*, which removes the table from the database. Do you need to modify the file manager also?

16.11 Revise the SimpleDB code so that statistics are stored in the catalog tables, and updated each time the database is changed.

16.12 Revise the SimpleDB code so that V(T, F) is computed for each table T and field F. (HINT: Keeping track of the count of each field can be memory-intensive, as the number of distinct values may be unbounded. A reasonable idea is to count values for a portion of the table, and extrapolate. For example, one might count how many records are required to read 1000 different values.)

16.13 Suppose that a client creates a table, inserts some records into it, and then does a rollback.
a) What happens to the table's metadata in the catalog?
b) What happens to the file containing the data? Explain what problem could occur if a client subsequently creates a table with the same name but a different schema.
c) Fix the SimpleDB code so that this problem is solved.

16.14 Modify the index manager so that it also saves the type of the index in the catalog. Assume that there are two types of index, in classes *BTreeIndex* and *HashIndex*. The class constructor and static method *searchCost* have the same arguments in each of these classes.

16.15 The SimpleDB index manager uses the table *idxcat* to hold index information. Another design possibility is to keep index information in the catalog table *fldcat*.
a) Compare the two possibilities. What are the advantages of each way?
b) Implement this alternative way.

16.16 Write a test driver for the *simpledb.metadata* package. Your driver should build upon the code fragment of Figure 16-2. It could create several new tables in a new database, get and print the table metadata, open up record chains for each table, insert some records, and get/set their field values, all using the metadata retrieved from the catalog.

17

QUERY PROCESSING

and the package simpledb.query

In the previous two chapters we saw how tables get implemented in files of records, and how the catalog holds the metadata needed to access these records. The next three chapters consider how the database system can support queries over these tables.

Chapter 4 examined two query languages: relational algebra, and SQL. Relational algebra turns out to be relatively easy to implement, because each operator denotes a small, well-defined task. SQL, on the other hand, is difficult to implement directly, because a single SQL query can embody several tasks. In fact, we saw in Chapter 4 how any SQL query can be expressed as a tree of relational algebra queries. This correspondence provides the key to implementing SQL – the database system first translates an SQL query to relational algebra, and then executes the relational algebra query tree.

This chapter examines the topic of how to execute relational algebra. The next two chapters examine how to translate SQL into relational algebra.

17.1 Scans

A sc*an* is an object that represents a relational algebra query. Scans in SimpleDB implement the interface *Scan*; see Figure 17-1. The *Scan* methods are essentially the same as in *RecordFile*. Clients iterate through a scan, moving from one output record to the next and retrieving field values. The scan manages the execution of the query, by moving appropriately through record files and comparing values.

```
public interface Scan {
   public void     beforeFirst();
   public boolean  next();
   public void     close();
   public Constant getVal(String fldname);
   public int      getInt(String fldname);
   public String   getString(String fldname);
   public boolean  hasField(String fldname);
}
```

Figure 17-1: The SimpleDB *Scan* interface

Figure 17-2 illustrates the use of a scan. The method *printNameAndGradYear* iterates through its input scan, printing the values of the fields *sname* and *gradyear* for each record. Since *Scan* is an interface, this function has no knowledge of how the scan was constructed or what query it represents. It works for any query whose output table contains a student name and a graduation year.

```
public static void printNameAndGradyear(Scan s) {
      while (s.next()) {
            String sname  = s.getString("sname");
            String gradyr = s.getInt("gradyear");
            System.out.println(sname + "\t" + gradyr);
      }
      s.close();
}
```

Figure 17-2: Printing the name and graduation year of each output record of a scan

The methods in *Scan* differ from those in *RecordFile* in three ways:

- A record file has methods to both read and modify the file, whereas a scan is read-only. Modification methods are defined only for *update scans*, as described in Section 17.2.
- A record file needs to know the entire schema of its records, but a scan does not. Instead, a scan only needs the method *hasField*, which tells whether a specified field is in the scan's output table.
- Scans have a method *getVal* in addition to *getInt* and *getString*. Method *getVal* returns an object of type *Constant*, which abstracts integers and strings (see Section 17.7). This method allows the query processor to examine and compare values from the database without knowing (or caring) what type they are.

Each *Scan* object corresponds to a node in a query tree. There is a *Scan* class for each relational operator; objects from those classes constitute the internal nodes of the query tree[†]. There is also a scan class for tables, whose objects correspond to the leaves of the tree. Figure 17-3 shows the scan constructors for tables and the three basic operators supported by SimpleDB.

```
public SelectScan(Scan s, Predicate pred);
public ProjectScan(Scan s, Collection<String> fldlist);
public ProductScan(Scan s1, Scan s2);
public TableScan(TableInfo ti, Transaction tx);
```

Figure 17-3: The API of the constructors in SimpleDB that implement *Scan*

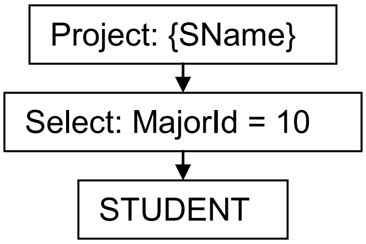
[†] In this chapter we assume that there is exactly one *Scan* class per relational operator. In Part 4, we allow operators to have multiple *Scan* classes.

Consider the *SelectScan* constructor, which takes two arguments: an underlying scan and a predicate. The underlying scan denotes the input table to the *select* operator. Recall that the input to a relational operator can be any table or query. This situation corresponds to having the input table be of type *Scan*. Since *Scan* is an interface, the *SelectScan* object is able to operate equally on its input table, whether it be a stored table or the output of another query.

The selection predicate passed into the *SelectScan* constructor is of type *Predicate*. Section 17.7 discusses the details of how SimpleDB handles selection predicates; until then, we shall remain somewhat vague on the issue.

A query tree can be represented by composing scans.
There will be a scan for each node of the tree.

Figures 17-4 and 17-5 give two examples of how query trees are implemented as scans.



(a) A query tree to retrieve the names of students having major #10

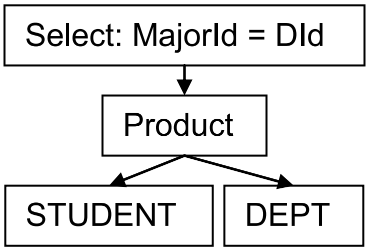
```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();

// the STUDENT node
TableInfo ti = SimpleDB.mdMgr().getTableInfo("student",tx);
Scan s1 = new TableScan(ti, tx);

// the Select node
Predicate pred = new Predicate(. . .); //majorid=10
Scan s2 = new SelectScan(s1, pred);

// the Project node
Collection<String> c = Arrays.asList("sname");
Scan s3 = new ProjectScan(s2, c);

while (s3.next())
      System.out.println(s3.getString("sname"));
s3.close();
```

(b) The SimpleDB code corresponding to the tree

Figure 17-4: Representing a query tree as a scan

Figure 17-4(a) depicts the tree for a query that returns the names of the students having major #10. Figure 17-4(b) gives the SimpleDB code that constructs and traverses this tree (omitting the details on the selection predicate). The *Scan* variables s1, s2, and s3 each correspond to a node in the query tree. The tree is built bottom-up: First the table scan is created, then the select scan, and finally the project scan. Variable s3 holds the final query tree. The while-loop traverses s3, printing each student name.

Figure 17-5(a) depicts the tree for a query that returns all information about students and their departments. Figure 17-5(b) gives the corresponding SimpleDB code, which is analogous to the previous example. In this case, the code contains four scans, because the query tree has four nodes. Variable s4 holds the final query tree. Note how the while-loop is nearly identical to the previous one. In the interest of saving space, the loop only prints three field values for each output record, but it can easily be modified to include all six field values.



(a) A query tree to join the STUDENT and DEPT tables

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();
MetadataMgr mdMgr = SimpleDB.mdMgr();

// the STUDENT node
TableInfo sti = mdMgr.getTableInfo("student", tx);
Scan s1 = new TableScan(sti, tx);

// the DEPT node
TableInfo dti = mdMgr.getTableInfo("dept", tx);
Scan s2 = new TableScan(dti, tx);

// the Product node
Scan s3 = new ProductScan(s1, s2);

// the Select node
Predicate pred = new Predicate(. . .); //majorid=did
Scan s4 = new SelectScan(s3, pred);

while (s4.next())
      System.out.println(s4.getString("sname")
                + ", " + s4.getString("gradyear")
                + ", " + s4.getString("dname") );
s4.close();
```

(b) The SimpleDB code corresponding to the tree

Figure 17-5: Representing another query tree as a scan

Finally, we note that the *close* method should be called only on the outermost scan of a query tree, as shown in Figures 17-4 and 17-5. Closing a scan automatically closes its underlying scans.

17.2 Update Scans

A query defines a virtual table. The *Scan* interface has methods that allow clients to read from this virtual table, but not to modify it. In Section 4.5.3 we considered the problem of updating a virtual table, and concluded that updates are meaningful only for tables defined by certain kinds of queries, which we called *updatable queries*. This same analysis applies to scans.

We say that a scan is *updatable* if every record *r* in the scan has a corresponding record *r′* in some underlying database table. In this case, an update to virtual record *r* is defined as an update to stored record *r′*. Updatable scans support the interface *UpdateScan*; see Figure 17-6. The first five methods of the interface correspond to the basic modification operations. The last two methods allow clients to obtain the rid of the single underlying record and to move to it. (Rids do not make sense for general scans, because a scan may have several underlying records.)

```
public interface UpdateScan extends Scan {
   public void setVal(String fldname, Constant val);
   public void setInt(String fldname, int val);
   public void setString(String fldname, String val);
   public void insert();
   public void delete();

   public RID  getRid();
   public void moveToRid(RID rid);
}
```

Figure 17-6: The SimpleDB *UpdateScan* interface

The only two classes in SimpleDB that implement *UpdateScan* are *TableScan* and *SelectScan*. As an example of their use, consider Figure 17-7. Figure 17-7(a) gives an SQL statement that changes the grade of every student who took section #53; Figure 17-7(b) gives the code that implements this statement. The code first creates a select scan of all enrollment records for section 53; it then iterates through the scan, changing the grade of each record.

```
update ENROLL
set Grade = 'C'
where SectionId = 53
```

(a) An SQL statement to modify the grades of students in section #53

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();

TableInfo ti = SimpleDB.mdMgr().getTableInfo("student",tx);
Scan s1 = new TableScan(ti, tx);

Predicate pred = new Predicate(. . .); //SectionId=53
UpdateScan s2  = new SelectScan(s1, pred);

while (s2.next())
      s2.setString("grade", "C");
s2.close();
```

(b) The SimpleDB code corresponding to the update

Figure 17-7: Representing an SQL update statement as an update scan

Note that variable s2 must be declared as an update scan, because it calls the method *setString*. On the other hand, the first argument to the *SelectScan* constructor is a scan, which means that s1 need not be declared as an update scan. Instead, the code for s2's *setString* method will cast its underlying scan (i.e. s1) to an update scan; if that scan is not updatable, a *ClassCastException* will be thrown.

17.3 Implementing Scans

The basic SimpleDB system contains four *Scan* classes: a classes for tables, and one class for the operators *select*, *project*, and *product*. The following subsections discuss these classes in detail.

17.3.1 Table scans

The code for class *TableScan* appears in Figure 17-8. Table scans are updatable, and thus implement *UpdateScan*. The code is straightforward. The constructor opens a record file for the table, using the *TableInfo* object that is passed in. The method *getVal* creates the appropriate constant, based on the type of the field. The method *setVal* performs the opposite action, extracting the value from the constant and sending it to the record file. The method *hasField* checks the table's schema. And the remaining methods are passed on to the record file.

```
public class TableScan implements UpdateScan {
   private RecordFile rf;
   private Schema sch;

   public TableScan(TableInfo ti, Transaction tx) {
      rf  = new RecordFile(ti, tx);
      sch = ti.schema();
   }

   // Scan methods

   public void beforeFirst() {
      rf.beforeFirst();
   }

   public boolean next() {
      return rf.next();
   }

   public void close() {
      rf.close();
   }

   public Constant getVal(String fldname) {
      if (sch.type(fldname) == INTEGER)
         return new IntConstant(rf.getInt(fldname));
      else
         return new StringConstant(rf.getString(fldname));
   }

   public int getInt(String fldname) {
      return rf.getInt(fldname);
   }

   public String getString(String fldname) {
      return rf.getString(fldname);
   }

   public boolean hasField(String fldname) {
      return sch.hasField(fldname);
   }

   // UpdateScan methods

   public void setVal(String fldname, Constant val) {
      if (sch.type(fldname) == INTEGER)
         rf.setInt(fldname, (Integer)val.asJavaVal());
      else
         rf.setString(fldname, (String)val.asJavaVal());
   }

   public void setInt(String fldname, int val) {
      rf.setInt(fldname, val);
   }

   public void setString(String fldname, String val) {
      rf.setString(fldname, val);
   }

   public void delete() {
      rf.delete();
   }

   public void insert() {
      rf.insert();
   }

    public RID getRid() {
      return rf.currentRid();
   }

    public void moveToRid(RID rid) {
      rf.moveToRid(rid);
   }
}
```

Figure 17-8: The code for the SimpleDB class *TableScan*

17.3.2 Select scans

The code for *SelectScan* appears in Figure 17-9. The constructor saves the scan of its underlying input table. A scan's current record is the same as the current record of its underlying scan, which means that most methods can be implemented by simply calling the corresponding methods of that scan.

```
public class SelectScan implements UpdateScan {
   private Scan s;
   private Predicate pred;

   public SelectScan(Scan s, Predicate pred) {
      this.s = s;
      this.pred = pred;
   }

   // Scan methods

   public void beforeFirst() {
      s.beforeFirst();
   }

   public boolean next() {
      while (s.next())
         if (pred.isSatisfied(s))
            return true;
      return false;
   }

   public void close() {
      s.close();
   }

   public Constant getVal(String fldname) {
      return s.getVal(fldname);
   }

   public int getInt(String fldname) {
      return s.getInt(fldname);
   }

   public String getString(String fldname) {
      return s.getString(fldname);
   }

   public boolean hasField(String fldname) {
      return s.hasField(fldname);
   }

   // UpdateScan methods

   public void setVal(String fldname, Constant val) {
      UpdateScan us = (UpdateScan) s;
      us.setVal(fldname, val);
   }

   public void setInt(String fldname, int val) {
      UpdateScan us = (UpdateScan) s;
      us.setInt(fldname, val);
   }

   public void setString(String fldname, String val) {
      UpdateScan us = (UpdateScan) s;
      us.setString(fldname, val);
   }

   public void delete() {
      UpdateScan us = (UpdateScan) s;
      us.delete();
   }

   public void insert() {
      UpdateScan us = (UpdateScan) s;
      us.insert();
   }

    public RID getRid() {
      UpdateScan us = (UpdateScan) s;
      return us.getRid();
   }

   public void moveToRid(RID rid) {
      UpdateScan us = (UpdateScan) s;
      us.moveToRid(rid);
   }
}
```

Figure 17-9: The code for the SimpleDB class *SelectScan*

The only nontrivial method is *next.* The job of this method is to establish a new current record. The code therefore loops through the underlying scan, looking for a record that satisfies the predicate. If such a record is found, then it becomes the current record and the method returns *true*. If there is no such record, then the while-loop will complete, and the method will return *false*.

Select scans are updatable. The *UpdateScan* methods assume that the underlying scan is also updatable; in particular, they assume that they can cast the underlying scan to *UpdateScan* without causing a *ClassCastException*. Since the SimpleDB query planner creates update scans that only involve table scans and select scans, an occurrence of such an exception would indicate a bug in the code.

17.3.3 Project scans

The code for *ProjectScan* appears in Figure 17-10. The list of output fields is passed into the constructor, which is used to implement method *hasField*. The other methods simply forward their requests to the corresponding method of the underlying schema. The *getVal*, *getInt*, and *getString* methods also check to see if the specified fieldname is in the field list; if not, an exception is generated.

```
public class ProjectScan implements Scan {
   private Scan s;
   private Collection<String> fieldlist;

   public ProjectScan(Scan s, Collection<String> fieldlist) {
      this.s = s;
      this.fieldlist = fieldlist;
   }

   public void beforeFirst() {
      s.beforeFirst();
   }

   public boolean next() {
      return s.next();
   }

   public void close() {
      s.close();
   }

   public Constant getVal(String fldname) {
      if (hasField(fldname))
         return s.getVal(fldname);
      else
         throw new RuntimeException("field not found.");
   }

   public int getInt(String fldname) {
      if (hasField(fldname))
         return s.getInt(fldname);
      else
         throw new RuntimeException("field not found.");
   }

   public String getString(String fldname) {
      if (hasField(fldname))
         return s.getString(fldname);
      else
         throw new RuntimeException("field not found.");
   }

   public boolean hasField(String fldname) {
      return fieldlist.contains(fldname);
   }
}
```

Figure 17-10: The code for the SimpleDB class *ProjectScan*

The class *ProjectScan* does not implement *UpdateScan*, even though projections are updatable. Exercise 17.16 asks you to complete the implementation.

17.3.4 Product Scans

The code for *ProductScan* appears in Figure 17-11.

```
public class ProductScan implements Scan {
   private Scan s1, s2;

   public ProductScan(Scan s1, Scan s2) {
      this.s1 = s1;
      this.s2 = s2;
      s1.next();
   }

   public void beforeFirst() {
      s1.beforeFirst();
      s1.next();
      s2.beforeFirst();
   }

   public boolean next() {
      if (s2.next())
         return true;
      else {
         s2.beforeFirst();
         return s2.next() && s1.next();
      }
   }

   public void close() {
      s1.close();
      s2.close();
   }

   public Constant getVal(String fldname) {
      if (s1.hasField(fldname))
         return s1.getVal(fldname);
      else
         return s2.getVal(fldname);
   }

   public int getInt(String fldname) {
      if (s1.hasField(fldname))
         return s1.getInt(fldname);
      else
         return s2.getInt(fldname);
   }

   public String getString(String fldname) {
      if (s1.hasField(fldname))
         return s1.getString(fldname);
      else
         return s2.getString(fldname);
   }

   public boolean hasField(String fldname) {
      return s1.hasField(fldname) || s2.hasField(fldname);
   }
}
```

Figure 17-11: The code for the SimpleDB class *ProductScan*

The scan needs to be able to iterate through all possible combinations of records from its underlying scans *s1* and *s2*. It does so by starting at the first record of *s1* and iterating through each record of *s2*, then moving to the second record of *s1* and iterating through *s2*, etc. Conceptually, it is like having a nested loop with the outer loop iterating *s1* and the inner loop iterating *s2*.

The method *next* implements this "nested loops" idea as follows. Each call to *next* moves to the next record of *s2*. If *s2* has such a record, then it can return *true*. If not, then the iteration of *s2* is complete, so the method moves to the next record of *s1* and the first record of *s2*. If this is possible, then it again returns *true*; if there are no more records of s1, then the scan is complete, and *next* returns *false*.

The *getVal*, *getInt*, and *getString* methods simply access the field of the appropriate underlying scan. Each method checks to see if the specified field is in scan *s1*. If so, then it accesses the field using *s1*; otherwise, it accesses the field using *s2*.

17.4 Pipelined Query Processing

We have just seen scan implementations for the three basic relational algebra operators of SimpleDB. These implementations are called *pipelined*, and have the following two characteristics:

> *Pipelined implementations have two characteristics:*
>
> - *They generate their output records one at a time, as needed.*
> - *They do not save their output records, nor do they save any intermediate computation.*

We shall analyze these characteristics for *TableScan* and *SelectScan* objects. The analysis of *ProductScan* and *ProjectScan* are similar.

Consider a *TableScan* object. It holds a record file, which holds a record page, which holds a buffer, which holds a page that contains the current record. The current record is just a location in that page. The record doesn't need to be removed from its page; if a client requests the value of a field, then the record manager will extract only that value, and send it back to the client. Each call to *next* moves to the next record in the record file, which may cause a new page to be read into the buffer.

Now consider a *SelectScan* object. Each call to its *next* method repeatedly calls *next* on its underlying scan, until the current record of the underlying scan satisfies the predicate. But of course there is no actual "current record" – If the underlying scan is a table scan, then (as we have seen) the current record is just some location in a page. And if the underlying scan is another kind of scan (such as the product scan in Figure 17-5), then the values of the current record are determined from the current records of the table scans that are in that node's subtree.

Each time the select scan processes another call to *next*, it starts its search from where it left off. As a result, the scan requests only as many records from its underlying scan as it needs in order to determine the next output record.

Finally, a select scan does not keep track of the records it has selected, which means that if the client asks for the records a second time, the select scan will need to perform the entire search all over again.

The term "pipelined" refers to the flow of the method calls down the query tree, and the flow of result values back up the tree. For example, consider a call to the method *getInt*. Each node in the tree passes that call down to one of its child nodes, until a leaf node is reached. That leaf node then obtains desired the value from the current record of its record file, and passes the value back up the tree. Or consider a call to the method *next*. Each node makes one or more calls to *next* (and possibly *beforeFirst*, in the case of a product node) on its child nodes, until it is satisfied that the underlying buffers contain the contents of the next record. It then returns success to its parent node (or failure, if no such record exists).

Pipelined implementations can be exceptionally efficient. For example, look at the query tree of Figure 17-12. We can show that the project and select nodes in this tree have zero cost.

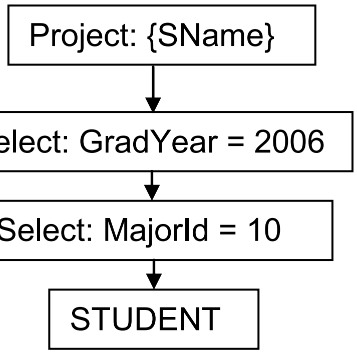


Figure 17-12: A query tree containing multiple *select* nodes

Consider first the project node. Each call to *next* on that node will simply call *next* on its child node, and pass back the return value of that node. In other words, the project node doesn't change the number of block accesses performed by the rest of the query.

Now consider the select nodes. A call to *next* on the outer select node will call *next* on the inner select node. The inner node will repeatedly call *next* on its child until the current record satisfies the predicate "MajorId=10". The inner select node then returns *true*, and the outer select node examines the current record. If its grad year is not 2006, then the outer node will call *next* on the inner node again, and await another current record. The only way for the outer select node to return *true* is if that record satisfies both predicates. This process continues each time the outer node calls *next*, with the underlying record file continually moving to its next record until both predicates are satisfied. When the record file recognizes that there are no more STUDENT records, its *next* method will return *false*, and the value of *false* will propagate up the tree. At this point, we can see that STUDENT is scanned only once, which is exactly the same as if we had executed just a table scan. Thus the select nodes in this query are also cost-free.

Although pipelined implementations are very efficient in some cases, there are other cases when they are not so good. Once such case is when a select node is on the right side of a product node, and therefore will get executed multiple times. Instead of performing the selection over and over, it may be better to use an implementation that materializes the output records and stores them in a temporary table. Such implementations are the topic of Chapter 22.

17.5 The Cost of Evaluating a Scan

One of the jobs of the database system is to construct the most cost-effective scan for a given SQL query. Consequently, the system needs to estimate how long it will take to iterate through a scan. Since the number of block accesses performed by a scan is by far the most important factor in determining running time, it suffices to estimate the number of block accesses required. In this section we consider how to estimate the cost of a scan. In Chapter 24 we will see how the database system makes use of this information.

The cost of a scan is estimated using terminology similar to the statistics of Section 16.4.

```
Let s be a scan, and let F be a field in the schema of s.

   B(s)    = the number of block accesses required to
             construct the output of s.
   R(s)    = the number of records in the output of s.
   V(s, F) = the number of different F-values in the
             output of s.
```

Figure 17-13 contains the cost formulas for each relational operator. These formulas are derived in the following subsections.

s	B(s)	R(s)	V(s,F)
TableScan(T)	B(T)	R(T)	V(T,F)
SelectScan(s_1,A=c)	B(s_1)	R(s_1)/V(s_1,A)	1 if F = A min{R(s), V(s_1,F)} if F ≠ A
SelectScan(s_1,A=B)	B(s_1)	R(s_1) / min{V(s_1,A), V(s_1,B)}	min{V(s_1,A), V(s_1,B)} if F = A,B min{R(s), V(s_1,F)} if F ≠ A,B
ProjectScan(s_1,L)	B(s_1)	R(s_1)	V(s_1,F)
ProductScan(s_1,s_2)	B(s_1) + R(s_1)*B(s_2)	R(s_1)*R(s_2)	V(s_1,F) if F is in s_1 V(s_2,F) if F is in s2

Figure 17-13: The statistical cost formulas for scans

17.5.1 The cost of a table scan

Each table scan holds a record file, which holds its current record page, which holds a buffer, which holds a page. When the records in that page have been read, its buffer is unpinned and the next page in the file takes its place. Thus a single pass through the table scan will access each block exactly once, using a single buffer at a time.

The values for B(s), R(s), and V(s, F) are therefore the number of blocks, records, and distinct values in the underlying table.

17.5.2 The cost of a select scan

A select scan has a single underlying scan, which we will call s_1. Each call to method *next* will cause the select scan to make one or more calls to *s_1.next*; the method will return *false* when the call to *s_1.next* returns *false*. Each call to *getInt*, *getString*, or *getVal* simply requests the field value from s_1, and requires no block accesses. Thus a select scan requires exactly the same number of block accesses as its underlying scan. That is:

$$B(s) = B(s_1)$$

The calculation of R(s) and V(s, F) depends on the selection predicate. As an example, we shall analyze the common cases where the selection predicate equates a field either to a constant or to another field.

Selection on a constant

Suppose that the predicate is of the form "*A = c*" for some field A. If we assume (as is common) that the values in A are equally distributed, then there will be $R(s_1)/V(s_1, A)$ records that match the predicate. That is:

$$R(s) = R(s_1) \ / \ V(s_1, A)$$

The equal-distribution assumption also implies that the values of the other fields are still equally distributed in the output. Thus we have:

$$\begin{aligned} &V(s, A) = 1 \\ &V(s, F) = \min\{R(s), V(s_1, F)\} \text{ for all other fields F} \end{aligned}$$

Selection on a field

Now suppose that the predicate is of the form "*A = B*" for fields A and B. In this case, it is reasonable to assume that the values in fields A and B are somehow related. In particular, we assume that if there are more B-values than A-values (that is, $V(s_1, A) < V(s_1, B)$) then every A-value appears somewhere in B; and if there are more A-values than B-values, the opposite is true. (This assumption is certainly true if there is a key-foreign key relationship between A and B.) So suppose that there are more B-values than A-values, and consider any record in s_1. Its A-value has a $1/V(s_1, B)$ chance of matching with its B-value. Similarly, if there are more A-values than B-values, then its B-value has a $1/V(s_1, A)$ chance of matching its A-value. Thus:

$$R(s) = R(s_1) \ / \ \max\{V(s_1, A), V(s_1, B)\}$$

The equal-distribution assumption also implies that the each A-value will be equally likely to match with a B-value. Thus we have:

$$\begin{aligned} &V(s, F) = \min(V(s_1, A), V(s_1, B)\} \text{ for F = A or B} \\ &V(s, F) = \min\{R(s), V(s_1, F)\} \text{ for all fields F other than A or B} \end{aligned}$$

17.5.3 The cost of a project scan

As with select scans, a project scan has a single underlying scan (called s_1), and requires no additional block accesses beyond those required by its underlying scan. Moreover, a projection operation does not change the number of records, nor does it change the values of any records. Thus:

$$\begin{aligned}
B(s) &= B(s_1) \\
R(s) &= R(s_1) \\
V(s, F) &= V(s_1, F) \text{ for all fields } F
\end{aligned}$$

17.5.4 The cost of a product scan

A product scan has two underlying scans, s_1 and s_2. Its output consists of all combinations of records from s_1 and s_2. As the scan is traversed, the underlying scan s_1 will be traversed once, and the underlying scan s_2 will be traversed once for each record of s_1. The following formulas follow:

$$\begin{aligned}
&B(s) = B(s_1) + (R(s_1) * B(s_2)) \\
&R(s) = R(s_1) * R(s_2) \\
&V(s, F) = V(s_1, F) \text{ or } V(s_2, F)\text{, depending on which table } F \text{ comes from.}
\end{aligned}$$

It is extremely interesting and important to realize that the formula for B(s) is not symmetric with respect to s_1 and s_2. That is, the statement

```
Scan s3 = new ProductScan(s1, s2);
```

can result in a different number of block accesses than the logically-equivalent statement

```
Scan s3 = new ProductScan(s2, s1);
```

How different can it be? Define

$$RPB(s) = R(s) / B(s)$$

That is, RPB(s) denotes the "records per block" of scan s – the average number of output records that result from each block access of s. We can then rewrite the above formula as:

$$B(s) = B(s_1) + (RPB(s_1) * B(s_1) * B(s_2))$$

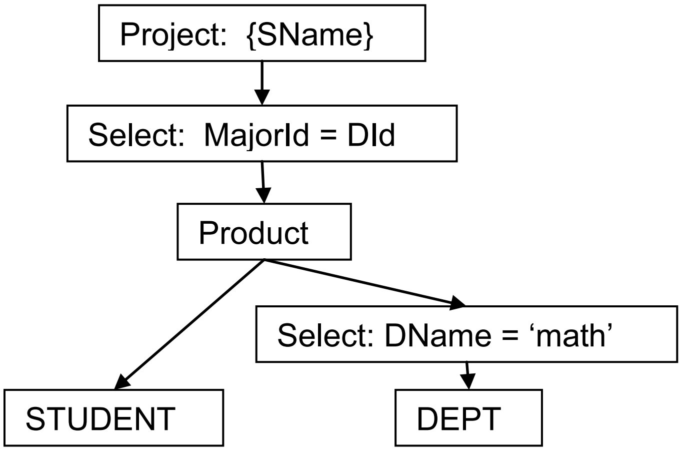
The dominating term is $(RPB(s_1) * B(s_1) * B(s_2))$. If we compare this term with the term we get by swapping s_1 with s_2, we see that the cost of the product scan will usually be lowest when s_1 is the underlying scan with the lowest RPB.

For example, suppose that s_1 is a table scan of STUDENT, and s_2 is a table scan of DEPT. Since STUDENT records are larger than DEPT records, more DEPT records fit in a block, which means that STUDENT has a smaller RPB than DEPT. The above analysis shows that the fewest number of disk accesses occur when the scan for STUDENT is first.

17.5.5 A concrete example

Consider the query that returns the names of those students majoring in math. Figure 17-14(a) depicts a query tree for that query, and Figure 17-14(b) gives the corresponding SimpleDB code. Figure 17-14(c) gives the statistics for these scans, assuming the statistics on the stored tables given in Figure 16-7.

(a)



(b)

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();
MetadataMgr mdMgr = SimpleDB.mdMgr();
TableInfo sti = mdMgr.getTableInfo("student", tx);
TableInfo dti = mdMgr.getTableInfo("dept", tx);
Scan s1 = new TableScan(sti, tx);
Scan s2 = new TableScan(dti, tx);
Predicate pred1 = new Predicate(. . .); //dname='math'
Scan s3 = new SelectScan(s2, pred1);
Scan s4 = new ProductScan(s1, s3);
Predicate pred2 = new Predicate(. . .); //majorid=did
Scan s5 = new SelectScan(s4, pred2);
Collection<String> fields = Arrays.asList("sname");
Scan s6 = new ProjectScan(s5, fields);
```

(c)

s	B(s)	R(s)	V(s,F)
s1	4,500	45,000	45,000 for F=SId 44,960 for F=SName 50 for F=GradYear 40 for F=MajorId
s2	2	40	40 for F=DId, DName
s3	2	1	1 for F=DId, DName
s4	94,500	45,000	45,000 for F=SId 44,960 for F=SName 50 for F=GradYear 40 for F=MajorId 1 for F=DId, DName
s5	94,500	1,125	1,125 for F=SId 1,124 for F=SName 50 for F=GradYear 1 for F=MajorId, DId, DName
s6	94,500	1,125	1,124 for F=SName

Figure 17-14: Calculating the cost of a scan

The entries for s1 and s2 simply duplicate the statistics for STUDENT and DEPT in Figure 16-7. The entry for s3 says that the selection on *DName* returns 1 record, but requires searching both blocks of DEPT to find it. Scan s4 returns all combinations of the 45,000 STUDENT records with the 1 selected record; the output is 45,000 records. However, the operation requires 94,500 block accesses, because the single math-department record must be found 45,000 times, and each time requires a 2-block scan of DEPT. (The other 4,500 block accesses come from the single scan of STUDENT.) The selection on *MajorId* in scan s5 reduces the output to 1,125 records (45,000 students / 40 departments), but does not change the number of block accesses required. And of course, the projection doesn't change anything.

It may seem strange that the database system recomputes the math-department record 45,000 times, and at considerable cost; however, this is the nature of pipelined query processing[†]. Looking at the RPB figures for STUDENT and s3, we see that RPB(STUDENT) = 10 and RPB(s3) = 0.5. Since products are fastest when the scan with the smaller RPB is on the left side, a more efficient strategy would be to define s4 as:

```
s4 = new ProductScan(s3, STUDENT)
```

Exercise 17.7 asks you to show that in this case, the operation would have required only 4,502 block accesses. The difference is due primarily to the fact that the selection is now computed only once.

17.6 Plans

An SQL query may have several equivalent query trees, with each tree corresponding to a different scan. The database planner is responsible for creating the scan having the lowest estimated cost. In order to do so, the planner may need to construct several query trees and compare their costs. However, it should create a scan only for the tree having lowest cost.

A query tree constructed for the purpose of cost comparison is called a plan.

Plans and scans are conceptually similar, in that they both denote a query tree. The difference is that a plan has methods for estimating costs; it accesses the database's metadata, but not the actual data. Multiple plans can be created without causing any disk-access overhead. A scan, on the other hand, is built for accessing the database. When a scan is created, a record file will be opened for each table mentioned in the query, which in turn will read the first block of its file into a buffer. Because opening a scan incurs disk accesses, a database system does not create a scan until it is sure that it is the best way to execute the query.

A plan implements the interface *Plan*; see Figure 17-15.

[†] This is a situation where the non-pipelined implementations of Chapter 22 are useful.

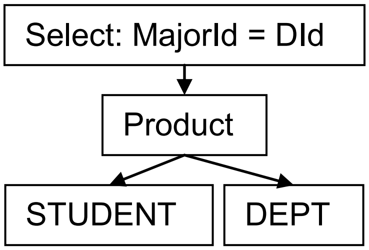
```
public interface Plan {
   public Scan   open();
   public int    blocksAccessed();
   public int    recordsOutput();
   public int    distinctValues(String fldname);
   public Schema schema();
}
```

Figure 17-15: The SimpleDB *Plan* interface

This interface supports the methods *blocksAccessed*, *recordsOutput*, and *distinctValues*, which allow the plan to calculate the values B(s), R(s), and V(s, F) for its query. The method *schema* returns the schema of the output table. The query planner can use this schema to verify type-correctness, and to look for ways to optimize the plan. Finally, every plan has the method *open*, which creates its corresponding scan.

A plan is constructed similarly to a scan. There is a *Plan* class for each relational algebra operator, and the class *TablePlan* handles stored tables. For example, consider Figure 17-16, which deals with the same query as Figure 17-4. The code fragments of Figure 17-4(b) and Figure 17-16(b) are similar; the main difference is that the latter creates a plan.

(a)



(b)

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();
Plan p1 = new TablePlan("student", tx);
Plan p2 = new TablePlan("dept", tx);
Plan p3 = new ProductPlan(p1, p2);
Predicate pred = new Predicate(. . .); //majorid=did
Plan p4 = new SelectPlan(p3, pred);

Scan s = p4.open();
while (s.next())
      System.out.println(s.getString("sname"));
s.close();
```

Figure 17-16: Creating and opening a query plan

Figures 17-17 through 17-20 give the code for classes *TablePlan*, *SelectPlan*, *ProjectPlan*, and *ProductPlan*. Class *TablePlan* obtains its cost estimates directly from *StatMgr*. The other classes use the formulas of Section 17.5 to compute their estimates.

```
public class TablePlan implements Plan {
   private Transaction tx;
   private TableInfo ti;
   private StatInfo si;

   public TablePlan(String tblname, Transaction tx) {
      this.tx = tx;
      ti = SimpleDB.mdMgr().getTableInfo(tblname, tx);
      si = SimpleDB.mdMgr().getStatInfo(tblname, ti, tx);
   }

   public Scan open() {
      return new TableScan(ti, tx);
   }

   public int blocksAccessed() {
      return si.blocksAccessed();
   }

   public int recordsOutput() {
      return si.recordsOutput();
   }

   public int distinctValues(String fldname) {
      return si.distinctValues(fldname);
   }

   public Schema schema() {
      return ti.schema();
   }
}
```

Figure 17-17: The code for the SimpleDB class *TablePlan*

Cost estimation for select plans is more complex than for the other operators, because the estimates depend on what the predicate is. A predicate therefore has methods *reductionFactor* and *equatesWithConstant* that the select plan can use. Method *reductionFactor* is used by *recordsAccessed* to calculate the extent to which the predicate reduces the size of the input table. Method *equatesWithConstant* is used by *distinctValues* to determine whether the predicate equates the specified field with a constant.

```
public class SelectPlan implements Plan {
   private Plan p;
   private Predicate pred;

   public SelectPlan(Plan p, Predicate pred) {
      this.p = p;
      this.pred = pred;
   }

   public Scan open() {
      Scan s = p.open();
      return new SelectScan(s, pred);
   }

   public int blocksAccessed() {
      return p.blocksAccessed();
   }

   public int recordsOutput() {
      return p.recordsOutput() / pred.reductionFactor(p);
   }

   public int distinctValues(String fldname) {
      if (pred.equatesWithConstant(fldname) != null)
         return 1;
      else
         return Math.min(p.distinctValues(fldname), recordsOutput());
   }

   public Schema schema() {
      return p.schema();
   }
}
```

Figure 17-18: The code for the SimpleDB class *SelectPlan*

The constructors of *ProjectPlan* and *ProductPlan* create their schemas from the schemas of their underlying plans. The *ProjectPlan* schema is created by looking up each field of the underlying field list, and adding that information to the new schema. The *ProductPlan* schema is the union of the underlying schemas.

```
public class ProjectPlan implements Plan {
   private Plan p;
   private Schema schema = new Schema();

   public ProjectPlan(Plan p, Collection<String> fieldlist) {
      this.p = p;
      for (String fldname : fieldlist)
         schema.add(fldname, p.schema());
   }

   public Scan open() {
      Scan s = p.open();
      return new ProjectScan(s, schema.fields());
   }

   public int blocksAccessed() {
      return p.blocksAccessed();
   }

   public int recordsOutput() {
      return p.recordsOutput();
   }

   public int distinctValues(String fldname) {
      return p.distinctValues(fldname);
   }

   public Schema schema() {
      return schema;
   }
}
```

Figure 17-19: The code for the SimpleDB class *ProjectPlan*

```
public class ProductPlan implements Plan {
   private Plan p1, p2;
   private Schema schema = new Schema();
   public ProductPlan(Plan p1, Plan p2) {
      this.p1 = p1;
      this.p2 = p2;
      schema.addAll(p1.schema());
      schema.addAll(p2.schema());
   }

   public Scan open() {
      Scan s1 = p1.open();
      Scan s2 = p2.open();
      return new ProductScan(s1, s2);
   }

   public int blocksAccessed() {
      return p1.blocksAccessed() +
            (p1.recordsOutput() * p2.blocksAccessed());
   }

   public int recordsOutput() {
      return p1.recordsOutput() * p2.recordsOutput();
   }

   public int distinctValues(String fldname) {
      if (p1.schema().hasField(fldname))
         return p1.distinctValues(fldname);
      else
         return p2.distinctValues(fldname);
   }

   public Schema schema() {
      return schema;
   }
}
```

Figure 17-20: The code for the SimpleDB class *ProductPlan*

The *open* method for each of these plan classes is straightforward. In general, constructing a scan from a plan has two steps:

- First, recursively construct the scan for each underlying plan.
- Second, pass those scans into the *Scan* constructor for the operator.

17.7 Predicates

17.7.1 Using Predicates

We end this chapter by examining the structure of predicates and how they are implemented and used in SimpleDB.

Structurally, we have the following definitions:

A predicate is a Boolean combination of terms.
A term is a comparison between two expressions.
An expression consists of operations on constants and field names.

For example, consider the following predicate:

```
        (GradYear = 2006 or GradYear = 2007) and MajorId = DId
```

This predicate consists of three terms. The first two terms compare the field name *GradYear* against a constant, and the third term compares two field names. Each expression in these terms involves no operations, and is either a field name or a constant. In general, an expression could have operators (such as arithmetic) or built-in functions.

In SimpleDB, an expression can only be a constant or a field name; a term can only compare expressions for equality; and a predicate can only create conjuncts of terms. We shall discuss this simplified case here. Exercises 17.11 to 17.13 ask you to consider the more general cases.

<u>*Predicate*</u>

```
   // used by the parser:
   public void      conjoinWith(Predicate pred);

   // used by a scan:
   public boolean   isSatisfied(Scan s);

   // used by a plan:
   public int       reductionFactor();

   // used by the query planner:
   public Predicate selectPred(Schema sch);
   public Predicate joinPred(Schema sch1, Schema sch2);
   public Constant  equatesWithConstant(String fldname);
   public String    equatesWithField(String fldname);
```

Figure 17-21: The API for SimpleDB predicates

Figure 17-21 gives its API for the class *Predicate*. These methods address the needs of several parts of the database system:

- The parser constructs the predicate as it processes the where clause of an SQL statement. It calls the method *conjoinWith* to conjoin another predicate to itself.
- A select scan evaluates the predicate by calling the method *isSatisfied*.
- A select plan estimates the selectivity of the predicate by calling the methods *reductionFactor* and *equatesWithConstant*.
- The query planner needs to analyze the scope of a predicate and to break it into smaller pieces. The method *selectPred* returns the portion of the predicate (if any) that is the selection predicate of a table having the specified schema.

Similarly, the method *joinPred* returns the join predicate (if any) of the two specified schemas. The method *equatesWithConstant* looks for a term of the form “A = c”, for some specified field A; the existence of such a term indicates that indexing is possible. The method *equatesWithField* looks for a term of the form “A=B”, for similar reasons.

The version of SQL used by SimpleDB supports two types of constant: integers and strings. Full SQL has many more constant types. For the most part, queries do not care about the types of their fields. In SimpleDB, only the *select* operator cares, and then only to ensure type compatibility. Consider for example the predicate *MajorId* = *DId*. The *select* operator does not care if these fields are integers or strings; the only thing that matters is whether the values can be meaningfully compared.

Query processing can therefore be simplified if the database system can pass around values without having to know their type. The SimpleDB interface *Constant* contains methods common to all constants (methods *equals*, *hashCode*, *toString*, and *compareTo*), as well as a method *asJavaVal* that converts the constant to its actual value in Java; see Figure 17-22.

```
public interface Constant extends Comparable<Constant> {
   public Object asJavaVal();
}
```

Figure 17-22: The SimpleDB *Constant* interface

The interface has an implementing class for each supported constant type. In SimpleDB, these classes are *IntConstant* and *StringConstant*. Figure 17-23 gives the code for *StringConstant*; the code for *IntConstant* is similar.

```
public class StringConstant implements Constant {
   private String val;

   public StringConstant(String s) {
      val = s;
   }

   public Object asJavaVal() {
      return val;
   }

   public boolean equals(Object obj) {
      StringConstant sc = (StringConstant) obj;
      return sc != null && val.equals(sc.val);
   }

   public int compareTo(Constant c) {
      StringConstant sc = (StringConstant) c;
      return val.compareTo(sc.val);
   }

   public int hashCode() {
      return val.hashCode();
   }

   public String toString() {
      return val;
   }
}
```

Figure 17-23: The code for the SimpleDB class *StringConstant*

Expressions in SimpleDB are implemented using the interface *Expression*; see Figure 17-24.

```
public interface Expression {
   public boolean  isConstant();
   public boolean  isFieldName();
   public Constant asConstant();
   public String   asFieldName();
   public Constant evaluate(Scan s);
   public boolean  appliesTo(Schema sch);
}
```

Figure 17-24: The SimpleDB *Expression* interface

This interface has two implementing classes, one for each kind of expression in SimpleDB; see Figures 17-25 and 17-26. The methods *isConstant*, *isFieldName*, *asConstant*, and *asFieldName* allow clients to get at the contents of the expression, and are used by the planner in analyzing a query. The method *appliesTo* tells the planner whether the expression mentions fields only in the specified schema. Method *evaluate* is used during query evaluation, and returns the value of the expression with respect to the current output record (as expressed by a scan). If the expression is a constant, then the scan is irrelevant, and the method simply returns the constant. If the expression is a field, then the method returns the field's value in the current record.

```
public class ConstantExpression implements Expression {
  private Constant val;

  public ConstantExpression(Constant c) {
    val = c;
  }

  public boolean isConstant() {
    return true;
  }

  public boolean isFieldName() {
    return false;
  }

  public Constant asConstant() {
    return val;
  }

  public String asFieldName() {
    throw new ClassCastException();
  }

  public Constant evaluate(Scan s) {
    return val;
  }

  public boolean appliesTo(Schema sch) {
    return true;
  }

  public String toString() {
    return val.toString();
  }
}
```

Figure 17-25: The code for the SimpleDB class *ConstantExpression*

```
public class FieldNameExpression implements Expression {
  private String fldname;

  public FieldNameExpression(String fldname) {
    this.fldname = fldname;
  }

  public boolean isConstant() {
    return false;
  }

  public boolean isFieldName() {
    return true;
  }

  public Constant asConstant() {
    throw new ClassCastException();
  }

  public String asFieldName() {
    return fldname;
  }

  public Constant evaluate(Scan s) {
    return s.getVal(fldname);
  }

  public boolean appliesTo(Schema sch) {
    return sch.hasField(fldname);
  }

  public String toString() {
    return fldname;
  }
}
```

Figure 17-26: The code for the SimpleDB class *FieldNameExpression*

The code for class *Term* appears in Figure 17-27. The method *reductionFactor* encodes the selection formulas of Section 17.5.2. The method has four cases: both sides of the term are fields, the left side only is a field, the right side only is a field, and both sides are constants.

```
public class Term {
  private Expression lhs, rhs;

  public Term(Expression lhs, Expression rhs) {
    this.lhs = lhs;
    this.rhs = rhs;
  }

  public int reductionFactor(Plan p) {
    String lhsName, rhsName;
    if (lhs.isFieldName() && rhs.isFieldName()) {
      lhsName = lhs.asFieldName();
      rhsName = rhs.asFieldName();
      return Math.max(p.distinctValues(lhsName),
                      p.distinctValues(rhsName));
    }
    if (lhs.isFieldName()) {
      lhsName = lhs.asFieldName();
      return p.distinctValues(lhsName);
    }
    if (rhs.isFieldName()) {
      rhsName = rhs.asFieldName();
      return p.distinctValues(rhsName);
    }
    // otherwise, the term equates constants
    if (lhs.asConstant().equals(rhs.asConstant()))
      return 1;
    else
      return Integer.MAX_VALUE;
  }

  public Constant equatesWithConstant(String fldname) {
    if (lhs.isFieldName() &&
         lhs.asFieldName().equals(fldname) &&
         rhs.isConstant())
          return rhs.asConstant();
    else if (rhs.isFieldName() &&
         rhs.asFieldName().equals(fldname) &&
      lhs.isConstant())
          return lhs.asConstant();
    else
          return null;
  }

  public String equatesWithField(String fldname) {
    if (lhs.isFieldName() &&
         lhs.asFieldName().equals(fldname) &&
         rhs.isFieldName())
          return rhs.asFieldName();
    else if (rhs.isFieldName() &&
         rhs.asFieldName().equals(fldname) &&
      lhs.isFieldName())
          return lhs.asFieldName();
    else
          return null;
  }
  public boolean appliesTo(Schema sch) {
    return lhs.appliesTo(sch) && rhs.appliesTo(sch);
  }

  public boolean isSatisfied(Scan s) {
    Constant lhsval = lhs.evaluate(s);
    Constant rhsval = rhs.evaluate(s);
    return rhsval.equals(lhsval);
  }

  public String toString() {
    return lhs.toString() + "=" + rhs.toString();
  }
}
```

Figure 17-27: The code for the SimpleDB class *Term*

The code for class *Predicate* appears in Figure 17-28. A predicate is implemented as a list of terms, and a predicate responds to its methods by calling corresponding methods on each of its terms.

```
public class Predicate {
   private List<Term> terms = new ArrayList<Term>();

   public Predicate() {}

   public Predicate(Term t) {
      terms.add(t);
   }

   public void conjoinWith(Predicate pred) {
      terms.addAll(pred.terms);
   }

   public boolean isSatisfied(Scan s) {
      for (Term t : terms)
         if (!t.isSatisfied(s))
            return false;
      return true;
   }

   public int reductionFactor(Plan p) {
      int factor = 1;
      for (Term t : terms)
         factor *= t.reductionFactor(p);
      return factor;
   }

   public Predicate selectPred(Schema sch) {
      Predicate result = new Predicate();
      for (Term t : terms)
         if (t.appliesTo(sch))
            result.terms.add(t);
      if (result.terms.size() == 0)
```

```
         return null;
      else
         return result;
   }

   public Predicate joinPred(Schema sch1, Schema sch2) {
      Predicate result = new Predicate();
      Schema newsch = new Schema();
      newsch.addAll(sch1);
      newsch.addAll(sch2);
      for (Term t : terms)
         if (!t.appliesTo(sch1)  &&
            !t.appliesTo(sch2) &&
             t.appliesTo(newsch))
               result.terms.add(t);
      if (result.terms.size() == 0)
         return null;
      else
         return result;
   }

   public Constant equatesWithConstant(String fldname) {
      for (Term t : terms) {
         Constant c = t.equatesWithConstant(fldname);
         if (c != null)
            return c;
      }
      return null;
   }

   public String equatesWithField(String fldname) {
      for (Term t : terms) {
         String s = t.equatesWithField(fldname);
         if (s != null)
            return s;
      }
      return null;
   }

   public String toString() {
      Iterator<Term> iter = terms.iterator();
      if (!iter.hasNext())
         return "";
      String result = iter.next().toString();
      while (iter.hasNext())
         result += " and " + iter.next().toString();
      return result;
   }
}
```

Figure 17-28: The code for the SimpleDB class *Predicate*

For an example, consider the following predicate:

```
SName = 'joe' and MajorId = DId
```

The code fragment of Figure 17-29 shows how this predicate would get created in SimpleDB.

```
Expression lhs1 = new FieldNameExpression("SName");
Expression rhs1 = new ConstantExpression(new StringConstant("joe"));
Term t1 = new Term(lhs1, rhs1);

Expression lhs2 = new FieldNameExpression("MajorId");
Expression rhs2 = new FieldNameExpression("DId");
Term t2 = new Term(lhs2, rhs2);

Predicate pred1 = new Predicate(t1);
Predicate pred2 = pred1.conjoinWith(new Predicate(t2));
```

Figure 17-29: SimpleDB code to create a predicate

Clearly, such code is tedious for people to write. However, the parser finds it natural to think of things this way, and since the parser is responsible for constructing the predicates, it gets to decide how to do it.

17.8 Chapter Summary

- A sc*an* is an object that represents a relational algebra query tree. Each relational operator has a corresponding class that implements the *Scan* interface; objects from those classes constitute the internal nodes of the query tree. There is also a scan class for tables, whose objects constitute the leaves of the tree.

- The *Scan* methods are essentially the same as in *RecordFile*. Clients iterate through a scan, moving from one output record to the next and retrieving field values. The scan manages the implementation of the query, by moving appropriately through record files and comparing values.

- A scan is *updatable* if every record *r* in the scan has a corresponding record *r′* in some underlying database table. In this case, an update to virtual record *r* is defined as an update to stored record *r′*.

- The methods of each scan class implement the intent of that operator. For example:
  - A select scan checks each record in its underlying scan, and returns only those that satisfy its predicate.
  - A product scan returns a record for every combination of records from its two underlying scans.
  - A table scan opens a record file for the specified table, which pins buffers and obtains locks as necessary.

- These scan implementations are called *pipelined* implementations. A pipelined implementation does not try to read ahead, cache, sort, or otherwise preprocess its data.

- A pipelined implementation does not construct output records. Each leaf in the query tree is a table scan, containing a buffer that holds the current record of that table. The "output record" of the operation is determined from the records in each buffer. Requests to get field values are directed down the tree to the appropriate table scan; results are returned from table scans back up to the root.

- Scans that use pipelined implementations operate on a need-to-know basis. Each scan will request only as many records from its children as it needs to determine its next record.

- In order to construct the most cost-effective scan for a given query, the database system needs to estimate the number of block accesses required to iterate through a scan. The following estimation functions are defined for a scan s:
    - B(s) denotes the number of block accesses required to iterate through s.
    - R(s) denotes the number of records output by s.
    - V(s,F) denotes the number of distinct F-values in the output of s.

- If s is a table scan, then these functions are equivalent to the statistical metadata for the table. Otherwise, each operator has a formula for computing the function based on the values of the functions on its input scans.

- An SQL query may have several equivalent query trees, with each tree corresponding to a different scan. The database planner is responsible for creating the scan having the lowest estimated cost. In order to do so, the planner may need to construct several query trees and compare their costs. It will create a scan only for the tree having lowest cost.

- A query tree constructed for the purpose of cost comparison is called a *plan*. Plans and scans are conceptually similar, in that they both denote a query tree. The difference is that a plan has methods for estimating costs; it accesses the database's metadata, but not the actual data. Creating a plan does not incur any disk accesses. The planner creates multiple plans and compares their costs. It then chooses the plan having the lowest cost and opens a plan from it.

17.9 Suggested Reading

The topic of pipelined query processing is a small piece of the query-processing puzzle, which also includes the topics of Part 4. The article [Graefe 1993] contains comprehensive information about query-processing techniques; Section 1 has a large discussion of scans, plans, and pipelined processing. The article [Chaudhuri 1998] discusses query trees, plans, and statistical gathering in addition to query optimization.

17.10 Exercises

CONCEPTUAL EXERCISES

17.1 Implement the following query as a scan, using Figure 17-5 as a template.

```
select sname, dname, grade
from STUDENT, DEPT, ENROLL, SECTION
where SId=StudentId and SectId=SectionId and DId=MajorId
and YearOffered=2005
```

17.2 Consider the following Java statements:

```
Plan p = new TablePlan("student", tx);
Scan s = p.open();
```

a) What locks will the transaction need to obtain in order to execute each of these statements?
b) Give scenarios that would cause the statement to wait for a lock, if possible.

17.3 Consider the code for *ProductScan*.
a) What problem can occur when the first underlying scan has no records? How should the code be fixed?
b) Explain why no problems occur when the second underlying scan has no records.

17.4 Suppose you want to take the product of STUDENT with STUDENT, in order to find all pairs of students.
a) One way is to create a single table plan on STUDENT, and use it twice in the product, as in the following code fragment:

```
Plan p = new TablePlan("student", tx);
Plan p2 = new ProductPlan(p, p);
Scan s = p2.open();
```

Explain why this will produce incorrect (and strange) behavior when the scan is executed.
b) A better way is to create two different table plans on STUDENT, and create a product plan on them. This returns all combinations of STUDENT records, but has a problem. What is it?

17.5 Consider the following relational algebra query:

```
T1 = select(DEPT, DName='math')
T2 = select(STUDENT, GradYear=2008)
product(T1, T2)
```

Using the same assumptions as in Section 17.5:
a) Calculate the number of disk accesses required to execute the operation.
b) Calculate the number of disk accesses required to execute the operation if the arguments to *product* are exchanged.

17.6 Calculate B(s), R(s), and V(s, F) for the queries of Figures 17-4, 17-5, and 17-12.

17.7 Show that if the arguments to the *product* operation in Section 17.5.5 were swapped, then the entire operation would require 4,502 block accesses.

17.8 In Section 17.5, we stated that the product of STUDENT and DEPT is more efficient when STUDENT is the outer scan. Using the statistics of Figure 16-7, calculate the number of block accesses the product would require.

PROGRAMMING EXERCISES

17.9 The *getVal*, *getInt*, and *getString* methods in *ProjectScan* check to make sure that argument field names are valid. None of the other scan classes do this. For each of the other scan classes:
a) Say what problem will occur (and in what method) if those methods are called with an invalid field.
b) Fix the SimpleDB code so that an appropriate exception is thrown.

17.10 Currently, SimpleDB supports only integer and string constants.
a) Revise SimpleDB to have other kinds of constant, such as short integers, byte arrays, and dates.
b) Exercise 12.17 asked you to modify the class *Page* to have get/set methods for types such as short integers, dates, etc. If you have done this exercise, add similar get/set methods to *Scan* and *UpdateScan* (and their various implementation classes), as well as to the record manager, the transaction manager and the buffer manager. Then modify the methods *getVal* and *setVal* appropriately.

17.11 Revise expressions to handle arithmetic operators on integers.

17.12 Revise the class *Term* to handle the comparison operators < and >.

17.13 Revise the class *Predicate* to handle arbitrary combinations of the boolean connectives *and*, *or* and *not*.

17.14 In Exercise 15.16 you extended the SimpleDB record manager to handle database null values. Now extend the query processor to handle nulls. In particular:

- Add a new class *NullConstant* that denotes a null.
- Modify the methods *getVal* and *setVal* in *TableScan* so that they recognize null values and handle them appropriately in the underlying record file.
- Determine which of the various *Expression*, *Term*, and *Predicate* classes need to be modified to handle null constants.

17.15 Implement plan and scan classes for each of the following relational operators discussed in Chapter 4.

a) Join
b) Rename
c) Extend
d) Union (do not remove duplicates)
e) Semijoin
f) Antijoin
g) Outerjoin (use the null values from problem 17.14)

17.16 Revise the class *ProjectScan* to be an update scan.

17.17 Suppose that *setString* is called with a string that is longer than is specified in the schema.
a) Explain what kinds of things can go wrong, and when they will be detected.
b) Fix the SimpleDB code so that the error is detected and handled appropriately.

17.18 Rewrite the class *TableMgr* (from Chapter 16) to use scans instead of record files. Is your code simpler this way?

17.19 Exercise 15.6 asked you to write methods *previous* and *afterLast* for class *RecordFile*.
a) Modify SimpleDB so that scans also have these methods.
b) Write a program to test your code. Note that you will not be able to test your changes on the SimpleDB server unless you also extend the JDBC methods. See Exercise 20.5.

17.20 Revise the SimpleDB server so that whenever a query is executed, the query and its corresponding plan are printed in the console window; this information will provide interesting insight into how the server is processing the query. There are two tasks required:
a) Revise all of the classes that implement the *Plan* interface so that they implement the method *toString*. This method should return a well-formatted string representation of the plan, similar to the relational algebra notation in this chapter.
b) Revise the method *executeQuery* (in class *simpledb.remote.RemoteStatementImpl*) so that it prints its input query and the string from part (a) in the server's console window.

18

PARSING

and the package simpledb.parse

A JDBC client submits an SQL statement to the server as a string. The database system must process this string to create a scan that can be executed. This process has two stages: a syntax-based stage, known as *parsing*; and a semantic-based stage, known as *planning*. We discuss parsing in this chapter, and planning in the next chapter.

18.1 Syntax vs. Semantics

> *The syntax of a language is a set of rules that describe the strings that could possibly be meaningful statements.*

For example, consider the following string:

```
select from tables T1 and T2 where b - 3
```

This string is not syntactically legal, for several reasons:

- The *select* clause must contain something.
- The identifier *tables* is not a keyword, and will be treated as a table name. But table names need to be separated by commas.
- The keyword *and* should separate predicates, not tables.
- The string "*b–3"* does not denote a predicate.

Each one of these problems causes this string to be completely meaningless as an SQL statement. There is no way that the database could ever figure out how to execute it, regardless of what the identifiers *tables, T1, T2*, and *b* happen to denote.

> *The semantics of a language specifies the actual meaning of a syntactically-correct string.*

On the other hand, consider the following syntactically-legal string:

```
select a from x, z where b = 3
```

We can infer that this statement is a query, which requests one field (named *a*) from two tables (named *x* and *z*), having predicate *b=3*. Thus the statement is possibly meaningful.

Whether the statement is actually meaningful depends on *semantic information* about *x, z, a,* and *b*. In particular, *x* and *z* must be names of tables, such that these tables contain a field named *a* and an integer-valued field named *b*. This semantic information can be determined from the database's metadata. The parser knows nothing about metadata, and thus cannot evaluate the meaningfulness of an SQL statement. Instead, the responsibility for examining the metadata belongs to the planner, as we shall see in Chapter 19.

18.2 Lexical Analysis

The first task of the parser is to split the input string into "chunks" called *tokens*. The portion of the parser that performs this task is called the *lexical analyzer*.

The lexical analyzer splits the input string into a series of tokens.

Each token has a *type* and a *value*. The SimpleDB lexical analyzer supports five token types:

- single-character *delimiters*, such as the comma;
- *integer constants*, such as *123*;
- *string constants*, such as *'joe'*;
- *keywords*, such as *select*, *from*, and *where*;
- *identifiers*, such as *STUDENT*, *x*, and *glop34a*.

Whitespace characters (spaces, tabs, and newlines) are generally not part of tokens; the only exception is inside of string constants. Whitespace is used to enhance readability and to separate tokens from each other.

Consider again the previous SQL statement:

```
select a from x, z where b = 3
```

The lexical analyzer creates the following 10 tokens for it:

TYPE	VALUE
keyword	select
identifier	a
keyword	from
identifier	x
delimiter	,
identifier	z
keyword	where
identifier	b
delimiter	=
intconstant	3

The parser calls the lexical analyzer when it needs to know about the token stream. There are two kinds of methods that the parser can call: methods that ask about the current token, and methods that tell the lexical analyzer to "eat" the current token, returning its value and moving to the next token. Each token type has a corresponding pair of methods. The API for these 10 methods are part of the SimpleDB class *Lexer*, and appear in Figure 18-1.

Lexer

```
public boolean matchDelim(char d);
public boolean matchIntConstant();
public boolean matchStringConstant();
public boolean matchKeyword(String w);
public boolean matchId();

public void    eatDelim(char d);
public int     eatIntConstant();
public String  eatStringConstant();
public void    eatKeyword(String w);
public String  eatId();
```

Figure 18-1: The API for the SimpleDB lexical analyzer

The first five methods return information about the current token. The method *matchDelim* returns *true* if the current token is a delimiter having the specified value. Similarly, *matchKeyword* returns *true* if the current token is a keyword having the specified value. The other three *matchXXX* methods return *true* if the current token is of the proper type.

The last five methods "eat" the current token. Each method calls its corresponding *matchXXX* method. If that method returns *false*, then an exception is thrown; otherwise, the next token becomes current. In addition, the methods *eatIntConstant*, *eatStringConstant*, and *eatId* return the value of the current token.

18.3 Implementing the Lexical Analyzer

Conceptually, the behavior of a lexical analyzer is straightforward – it reads the input string one character at a time, stopping when it determines that the next token has been read. The complexity of a lexical analyzer is in direct proportion to the set of token types: the more token types to look for, the more complex the implementation.

Java supplies two different built-in *tokenizers* (their term for lexical analyzers): one in class *StringTokenizer*, and one in class *StreamTokenizer*. The string tokenizer is simpler to use, but it only supports two kinds of token: delimiters and words (which are the substrings between delimiters). This is not appropriate for SQL, in particular because the string tokenizer does not understand numbers or quoted strings. On the other hand, the stream tokenizer has an extensive set of token types, including support for all five types used by SimpleDB. Thus the SimpleDB class *Lexer* is based on Java's stream tokenizer. Its code appears in Figure 18-2.

```
public class Lexer {
   private Collection<String> keywords;
   private StreamTokenizer tok;

   public Lexer(String s) {
      initKeywords();
      tok = new StreamTokenizer(new StringReader(s));
      tok.ordinaryChar('.');
      tok.lowerCaseMode(true); //converts ids and keywords to LowerCase
      nextToken();
   }

//Methods to check the status of the current token

   public boolean matchDelim(char d) {
      return d == (char)tok.ttype;
   }

   public boolean matchIntConstant() {
      return tok.ttype == StreamTokenizer.TT_NUMBER;
   }

   public boolean matchStringConstant() {
      return '\'' == (char)tok.ttype;
   }

   public boolean matchKeyword(String w) {
     return tok.ttype == StreamTokenizer.TT_WORD && tok.sval.equals(w);
   }

   public boolean matchId() {
     return tok.ttype == StreamTokenizer.TT_WORD &&
                         !keywords.contains(tok.sval);
   }

//Methods to "eat" the current token

   public void eatDelim(char d) {
      if (!matchDelim(d))
         throw new BadSyntaxException();
      nextToken();
   }

   public int eatIntConstant() {
      if (!matchIntConstant())
         throw new BadSyntaxException();
      int i = (int) tok.nval;
      nextToken();
      return i;
   }
   public String eatStringConstant() {
      if (!matchStringConstant())
         throw new BadSyntaxException();
      String s = tok.sval; //constants are not converted to lower case
      nextToken();
      return s;
   }

   public void eatKeyword(String w) {
      if (!matchKeyword(w))
         throw new BadSyntaxException();
      nextToken();
   }

   public String eatId() {
      if (!matchId())
         throw new BadSyntaxException();
      String s = tok.sval;
      nextToken();
      return s;
   }

   private void nextToken() {
      try {
         tok.nextToken();
      }
      catch(IOException e) {
         throw new BadSyntaxException();
      }
   }

   private void initKeywords() {
      keywords = Arrays.asList("select", "from", "where", "and",
                               "insert", "into", "values", "delete",
                               "update", "set", "create", "table",
                        "varchar", "int", "view", "as", "index", "on");
   }
}
```

Figure 18-2: The code for the SimpleDB class *Lexer*

The constructor for *Lexer* sets up the stream tokenizer. The call *tok.ordinaryChar('.')* tells the tokenizer to interpret the period character as a delimiter. (Even though periods are not used in SimpleDB, it is important to identify them as delimiters, so that they are not accepted as part of a field name or table name.) The call *tok.lowerCaseMode(true)* tells the tokenizer to convert all string tokens (but not quoted strings) to lower case, which allows SQL to be case-insensitive for keywords and identifiers.

The stream tokenizer keeps a public instance variable, called *ttype*, that holds the type of the current token; the lexical analyzer's *matchXXX* methods simply examine this value. Perhaps the one non-obvious method in the code is *matchStringConstant*, where the value of *ttype* is the quote character.

The methods *eatIntConstant*, *eatStringConstant*, and *eatId* return the value of the token. Numeric values are kept in the instance variable *tok.nval*; string values are kept in the variable *tok.sval*.

The Java method *StreamTokenizer.nextToken* throws an *IOException*. The private method *nextToken* transforms this exception to a *BadSyntaxException*, which is passed back to the client (and turned into an *SQLException*, as we shall see in Chapter 20).

The method *initKeywords* constructs a collection of all keywords used in SimpleDB's version of SQL.

18.4 Grammars

> *A grammar is a set of rules*
> *that describe how tokens can be legally combined.*

The following is an example of a grammar rule:

```
<Field> := IdTok
```

A grammar rule specifies the contents of the *syntactic category* named in its left-hand side. A syntactic category denotes a particular concept in the language. In the above rule, <*Field*> is intended to denote field names.

The *contents* of a syntactic category is the set of strings that correspond to its intended concept, and is specified by the pattern in the rule's right-hand side. In the above rule, the pattern is simply "*IdTok*". The term *IdTok* matches any identifier token, and so <Field> contains the set of strings corresponding to identifiers.

Each syntactic category can be thought of as its own mini-language. A string is a member of this language if it belongs to the category's contents. For example, the following two strings are members of <Field>:

```
SName
Glop
```

Remember that the identifiers need not be meaningful – they just need to be identifiers. So "Glop" is a perfectly good member of <Field>, even in the SimpleDB university database. However, the string "select" would not be a member of <Field>, because *select* is a keyword token, not an identifier token.

The right-hand side of a grammar rule can contain references to tokens and syntactic categories. Tokens that have well-known values (that is, keywords and delimiters) are written directly in the rule. Other tokens (identifiers, integer constants, and string constants) are written as *IdTok*, *IntTok*, and *StrTok* respectively. We also use three meta-characters ('[', ']', and '|') as punctuation; these characters are not delimiters in the language, so we put them to use to help express patterns. To illustrate, consider the following four additional grammar rules:

```
<Constant>   := StrTok | IntTok
<Expression> := <Field> | <Constant>
<Term>       := <Expression> = <Expression>
<Predicate>  := <Term> [ AND <Predicate> ]
```

The first rule defines the category <Constant>, which stands for any constant – string or integer. The meta-character ‘|’ means “or”; that is, the category <Constant> matches either string tokens or integer tokens, and its contents (as a language) will contain all string constants as well as all integer constants.

The second rule defines the category <Expression>, which denotes operator-free expressions. The rule specifies that an expression is either a field or a constant.

The third rule defines the category <Term>, which denotes simple equality terms between expressions (as in the SimpleDB class *Term*). For example, the following strings belong to <Term>:

```
DeptId = DId
'math' = DName
SName  = 123
65     = 'abc'
```

Recall that the parser does not check for type consistency; thus the last two strings are syntactically correct even though they are semantically incorrect.

The fourth rule defines the category <Predicate>, which stands for a Boolean combination of terms, similar to the SimpleDB class *Predicate*. The meta-characters ‘[‘ and ‘]’ are used to denote something optional. Thus the right-hand side of the rule matches any sequence of tokens that is either a <Term>, or a <Term> followed by an AND keyword token followed (recursively) by another <Predicate>. For example, the following strings belong to <Predicate>:

```
DName = 'math'
Id = 3 AND DName = 'math'
MajorId = DId AND Id = 3 AND DName = 'math'
```

The first string is of the form *<Term>*. The second two strings are of the form *<Term> AND <Predicate>*.

If a string belongs to a particular syntactic category, we can draw a *parse tree* to depict why. A parse tree has syntactic categories as its internal nodes, and tokens as its leaf nodes. The children of a category node correspond to the application of a grammar rule. For example, Figure 18-3 contains the parse tree for the string

```
DName = 'math' AND GradYear = SName
```

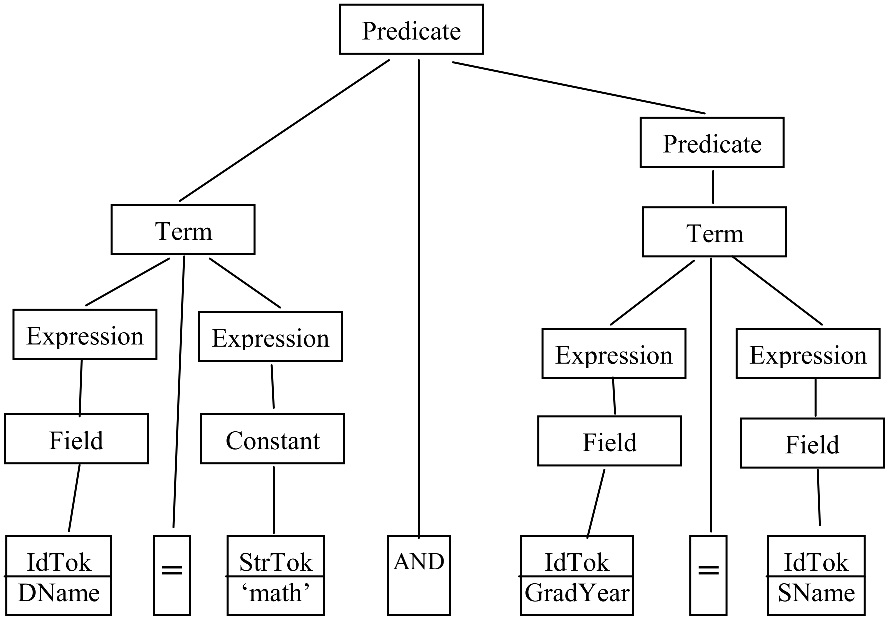


Figure 18-3: A parse tree for the string `DName = 'math' AND GradYear = SName`

In this figure, the tree's leaf nodes appear along the bottom of the tree (which allows you to read the input string). Starting from the root node, this tree asserts that the entire string is a <Predicate> because *"DName='math'"* is a <Term> and *"GradYear=SName"* is a <Predicate>. One can expand each of the subtrees similarly. For example, *"DName='math'"* is a <Term> because both *"DName"* and *" 'math' "* belong to <Expression>.

Figure 18-4 lists the entire grammar for the subset of SQL supported by SimpleDB. The grammar rules are divided into nine sections: one section for common constructs such as predicates, expressions, and fields, one section for queries, and seven sections for the various kinds of update statement.

Lists of items appear frequently in SQL. For example in a query, the *select* clause contains a comma-separated list of fields, the *from* clause contains a comma-separated list of identifiers, and the *where* clause contains an AND-separated list of terms. Each list is specified in the grammar using the same recursive technique that we saw for <Predicate>. Also note how the "option-bracket" notation is used in the rules for <Query>, <Delete>, and <Modify>, to allow them to have optional *where* clauses.

```
<Field>      := IdTok
<Constant>   := StrTok | IntTok
<Expression> := <Field> | <Constant>
<Term>       := <Expression> = <Expression>
<Predicate>  := <Term> [ AND <Predicate> ]

<Query>      := SELECT <SelectList> FROM <TableList> [ WHERE <Predicate> ]
<SelectList> := <Field> [ , <SelectList> ]
<TableList>  := IdTok [ , <TableList> ]

<UpdateCmd>  := <Insert> | <Delete> | <Modify> | <Create>
<Create>     := <CreateTable> | <CreateView> | <CreateIndex>

<Insert>     := INSERT INTO IdTok ( <FieldList> ) VALUES ( <ConstList> )
<FieldList>  := <Field> [ , <FieldList> ]
<ConstList>  := <Constant> [ , <Constant> ]

<Delete>     := DELETE FROM IdTok [ WHERE <Predicate> ]

<Modify>  := UPDATE IdTok SET <Field> = <Expression> [ WHERE <Predicate> ]

<CreateTable> := CREATE TABLE IdTok ( <FieldDefs> )
<FieldDefs>   := <FieldDef> [ , <FieldDefs> ]
<FieldDef>    := IdTok <TypeDef>
<TypeDef>     := INT | VARCHAR ( IntTok )

<CreateView>  := CREATE VIEW IdTok AS <Query>

<CreateIndex> := CREATE INDEX IdTok ON IdTok ( <Field> )
```

Figure 18-4: The grammar for the SimpleDB subset of SQL

We have seen that the parser cannot enforce type compatibility, because it cannot know the types of the identifiers it sees. The parser also cannot enforce compatible list sizes. For example, an SQL *insert* statement must mention the same number of values as field names, but the grammar rule for <Insertion> requires only that the string have an <IdList> and a <ConstList>. The planner must be responsible for verifying that these lists are the same size (and are type compatible)[†].

18.5 Recursive-Descent Parsers

A parse tree can be thought of as a proof that a given string is syntactically legal. But how does one determine the parse tree? How can the database system determine if a string is syntactically legal?

Programming-language researchers have developed numerous parsing algorithms for this purpose. The complexity of the parsing algorithm is usually in proportion to the complexity of the grammars it can support. Fortunately for us, our SQL grammar is

[†] This situation is certainly not desirable; in fact, it would be really nice to have a grammar in which equal list sizes are enforced. However, one can use automata theory to prove that no such grammar is possible.

about as simple as you can get, and so we will use the simplest possible parsing algorithm, known as *recursive descent*.

In a basic recursive descent parser, each syntactic category is implemented by a *void* method. A call to this method will “eat” those tokens that comprise the parse tree for that category, and return. The method throws an exception when the tokens do not correspond to a parse tree for that category.

For example, consider the first five grammar rules from Figure 18-4, which form the subset of SQL corresponding to predicates. A Java class corresponding to this grammar appears in Figure 18-5.

```
public class PredParser {
   private Lexer lex;

   public PredParser(String s) {
      lex = new Lexer(s);
   }

   public void field() {
      lex.eatId();
   }

   public void constant() {
      if (lex.matchStringConstant())
         lex.eatStringConstant();
      else
         lex.eatIntConstant();
   }

   public void expression() {
      if (lex.matchId())
         field();
      else
         constant();
   }

   public void term() {
      expression();
      lex.eatDelim('=');
      expression();
   }

   public void predicate() {
      term();
      if (lex.matchKeyword("and")) {
         lex.keyword("and");
         predicate();
      }
   }
}
```

Figure 18-5: The code for a simplified recursive-descent parser for predicates

A recursive-descent parser has a method for each grammar rule. Each method calls the methods of the items in the RHS of its rule.

For example, consider the method *field*, which makes one call to the lexical analyzer (and ignores any return values). If the next token is an identifier, then the call returns successfully, and that token will be eaten. If not, then the method throws the exception back to the caller. Similarly, consider the method *term.* Its first call to *expression* eats the tokens corresponding to a single SQL expression; its call to *eatDelim* eats the equals-sign token; and its second call to *expression* eats the tokens corresponding to another SQL expression. If any of these method calls did not find the tokens it expected, it would throw an exception, which the *term* method would pass on to its caller.

Grammar rules that contain alternatives are implemented using *if* statements. The condition of the *if* statement looks at the current token in order to decide what to do. For a trivial example, consider the method *constant*. If the current token is a string constant, then the method eats it; otherwise, the method tries to eat an integer constant. If the current token is neither a string constant nor an integer constant, then the call to *lex.eatIntConstant* will generate the exception. For a less trivial example, consider the method *expression*. Here the method knows that if the current token is an identifier, then it must look for a field; otherwise it must look for a constant[†].

The method *predicate* illustrates how a recursive rule is implemented. It first calls the method *term*, and then checks to see if the current token is the keyword *AND*. If so, it eats the *AND*-token and calls itself recursively; if the current token is not an *AND*, then it knows it has seen the last term in the list, and returns. Note that a call to *predicate* will eat as many tokens as it can from the token stream – if it sees an *AND*-token, it keeps going, even if it has already seen a valid predicate.

The interesting thing about recursive-descent parsing is that the sequence of method calls determines the parse tree for the input string. Exercise 18.4 asks you to modify the code of each method to print its name, appropriately indented; the result will resemble a sideways parse tree.

18.6 Adding Actions to the Parser

The basic recursive-descent parsing algorithm doesn't do anything, except return normally if the input string is syntactically valid. Although this behavior is somewhat interesting, it is not very useful if we want to actually execute an SQL statement. Consequently, we must modify the basic parser so that it returns information that the planner needs. This modification is called adding *actions* to the parser.

[†] This method also shows us the limitations of recursive descent parsing. If a grammar rule has two alternatives that both require the same first token, then there would be no way to know which alternative to take, and recursive descent will not work. In fact, you may have noticed that our grammar of Figure 18-4 has this very problem. See Exercise 18.3.

In general, the SQL parser should extract information such as table names, field names, predicates, and constants from the SQL statement. What gets extracted depends on the kind of SQL statement it is.

- *For a query*: a collection of field names (from the *select* clause), a collection of table names (from the *from* clause), and a predicate (from the *where* clause).
- *For an insertion*: a table name, a list of field names, and a list of values.
- *For a deletion*: a table name and a predicate.
- *For a modification*: a table name, the name of the field to be modified, an expression denoting the new field value, and a predicate.
- *For a table creation*: a table name and its schema.
- *For a view creation*: a table name and its definition.
- *For index creation*: an index name, a table name, and the name of the indexed field.

All of this information is extracted from the token stream, via the return values of the *Lexer* methods. Thus the strategy for modifying each parser method is straightforward: Get the return values from calls to *eatId*, *eatStringConstant*, and *eatIntConstant*; assemble them into an appropriate object; and return the object to the method's caller.

Figure 18-6 gives the code for the class *Parser*, whose methods implement the grammar of Figure 18-4. The following subsections examine this code in detail.

```
public class Parser {
   private Lexer lex;

   public Parser(String s) {
      lex = new Lexer(s);
   }

// Methods for parsing predicates, terms, expressions, etc.

   public String field() {
      return lex.eatId();
   }

   public Constant constant() {
      if (lex.matchStringConstant())
         return new StringConstant(lex.eatStringConstant());
      else
         return new IntConstant(lex.eatIntConstant());
   }

   public Expression expression() {
      if (lex.matchId())
         return new FieldNameExpression(field());
      else
         return new ConstantExpression(constant());
   }

   public Term term() {
      Expression lhs = expression();
      lex.eatDelim('=');
      Expression rhs = expression();
      return new Term(lhs, rhs);
   }

   public Predicate predicate() {
      Predicate pred = new Predicate(term());
      if (lex.matchKeyword("and")) {
         lex.eatKeyword("and");
         pred.conjoinWith(predicate());
      }
      return pred;
   }

// Methods for parsing queries

   public QueryData query() {
      lex.eatKeyword("select");
      Collection<String> fields = selectList();
      lex.eatKeyword("from");
      Collection<String> tables = tableList();
      Predicate pred = new Predicate();
      if (lex.matchKeyword("where")) {
         lex.eatKeyword("where");
         pred = predicate();
      }
      return new QueryData(fields, tables, pred);
   }

   private Collection<String> selectList() {
      Collection<String> L = new ArrayList<String>();
      L.add(field());
      if (lex.matchDelim(',')) {
         lex.eatDelim(',');
         L.addAll(selectList());
      }
      return L;
   }

   private Collection<String> tableList() {
      Collection<String> L = new ArrayList<String>();
      L.add(lex.eatId());
      if (lex.matchDelim(',')) {
         lex.eatDelim(',');
         L.addAll(tableList());
      }
      return L;
   }

// Methods for parsing the various update commands

   public Object updateCmd() {
      if (lex.matchKeyword("insert"))
         return insert();
      else if (lex.matchKeyword("delete"))
         return delete();
      else if (lex.matchKeyword("update"))
```

```
         return modify();
      else
         return create();
   }

   private Object create() {
      lex.eatKeyword("create");
      if (lex.matchKeyword("table"))
         return createTable();
      else if (lex.matchKeyword("view"))
         return createView();
      else
         return createIndex();
   }

// Method for parsing delete commands

   public DeleteData delete() {
      lex.eatKeyword("delete");
      lex.eatKeyword("from");
      String tblname = lex.eatId();
      Predicate pred = new Predicate();
      if (lex.matchKeyword("where")) {
         lex.eatKeyword("where");
         pred = predicate();
      }
      return new DeleteData(tblname, pred);
   }

// Methods for parsing insert commands

   public InsertData insert() {
      lex.eatKeyword("insert");
      lex.eatKeyword("into");
      String tblname = lex.eatId();
      lex.eatDelim('(');
      List<String> flds = fieldList();
      lex.eatDelim(')');
      lex.eatKeyword("values");
      lex.eatDelim('(');
      List<Constant> vals = constList();
      lex.eatDelim(')');
      return new InsertData(tblname, flds, vals);
   }

   private List<String> fieldList() {
      List<String> L = new ArrayList<String>();
      L.add(field());
      if (lex.matchDelim(',')) {
         lex.eatDelim(',');
         L.addAll(fieldList());
      }
      return L;
   }

   private List<Constant> constList() {
      List<Constant> L = new ArrayList<Constant>();
      L.add(constant());
      if (lex.matchDelim(',')) {
         lex.eatDelim(',');
         List<Constant> L2 = constList();
         L.addAll(L2);
      }
      return L;
   }

// Method for parsing modify commands

   public ModifyData modify() {
      lex.eatKeyword("update");
      String tblname = lex.eatId();
      lex.eatKeyword("set");
      String fldname = field();
      lex.eatDelim('=');
      Expression newval = expression();
      Predicate pred = new Predicate();
      if (lex.matchKeyword("where")) {
         lex.eatKeyword("where");
         pred = predicate();
      }
      return new ModifyData(tblname, fldname, newval, pred);
   }

// Methods for parsing create table commands

   public CreateTableData createTable() {
      lex.eatKeyword("table");
      String tblname = lex.eatId();
      lex.eatDelim('(');
      Schema sch = fieldDefs();
      lex.eatDelim(')');
      return new CreateTableData(tblname, sch);
   }

   private Schema fieldDefs() {
      Schema schema = fieldDef();
      if (lex.matchDelim(',')) {
         lex.eatDelim(',');
         Schema schema2 = fieldDefs();
         schema.addAll(schema2);
      }
      return schema;
   }

   private Schema fieldDef() {
      String fldname = field();
      return fieldType(fldname);
   }

   private Schema fieldType(String fldname) {
      Schema schema = new Schema();
      if (lex.matchKeyword("int")) {
         lex.eatKeyword("int");
         schema.addIntField(fldname);
      }
      else {
         lex.eatKeyword("varchar");
         lex.eatDelim('(');
         int strLen = lex.eatIntConstant();
         lex.eatDelim(')');
         schema.addStringField(fldname, strLen);
      }
      return schema;
   }

// Method for parsing create view commands

   public CreateViewData createView() {
      lex.eatKeyword("view");
      String viewname = lex.eatId();
      lex.eatKeyword("as");
      QueryData qd = query();
      return new CreateViewData(viewname, qd);
   }


//  Method for parsing create index commands

   public CreateIndexData createIndex() {
      lex.eatKeyword("index");
      String indexname = lex.eatId();
      lex.eatKeyword("on");
      String tblname = lex.eatId();
      lex.eatDelim('(');
      String fldname = field();
      lex.eatDelim(')');
      return new CreateIndexData(indexname, tblname, fldname);
   }
}
```

Figure 18-6: The code for the SimpleDB class *Parser*

18.6.1 Parsing Predicates and Expressions

The heart of the parser deals with the five grammar rules that define predicates and expressions, because they are used to parse several different kinds of SQL statement. Those methods in *Parser* are the same as in *PredParser* (in Figure 18-5), except that they now contain actions and return values. In particular, the method *field* grabs the fieldname from the current token, and returns it. The methods *constant*, *expression, term*, and *predicate* are similar, returning a *Constant* object, an *Expression* object, a *Term* object, and a *Predicate* object.

18.6.2 Parsing Queries

The method *query* implements the syntactic category <Query>. As it parses a query, it acquires the three items needed by the planner – the field names, the table names, and the predicate – and saves them in a *QueryData* object. The class *QueryData* makes these values available via its accessor methods *fields*, *tables*, and *pred*; see Figure 18-7. The class also has a method *toString*, which re-creates the query string. This method is needed when processing view definitions, as we shall see.

```
public class QueryData {
   private Collection<String> fields;
   private Collection<String> tables;
   private Predicate pred;

   public QueryData(Collection<String> fields,
                    Collection<String> tables, Predicate pred) {
      this.fields = fields;
      this.tables = tables;
      this.pred = pred;
   }


   public Collection<String> fields() {
      return fields;
   }

   public Collection<String> tables() {
      return tables;
   }

   public Predicate pred() {
      return pred;
   }

   public String toString() {
      String result = "select ";
      for (String fldname : fields)
         result += fldname + ", ";

      //remove final comma
      result = result.substring(0, result.length()-2);
      result += " from ";
      for (String tblname : tables)
         result += tblname + ", ";
      result = result.substring(0, result.length()-2);
      String predstring = pred.toString();
      if (!predstring.equals(""))
         result += " where " + predstring;
      return result;
   }
}
```

Figure 18-7: The code for the SimpleDB class *QueryData*

18.6.3 Parsing Updates

The parser method *updateCmd* implements the syntactic category <UpdateCmd>, which denotes the union of the various SQL update statements. This method will be called during the execution of the JDBC method *executeUpdate*, in order to determine which kind of update the command denotes. The method identifies the command from the initial tokens of the string, and then dispatches to the particular parser method for that command. Each update method has a different return type, because each one extracts different information from its command string; thus the method *updateCmd* returns a value of type *Object*.

18.6.4 Parsing Insertions

The parser method *insert* implements the syntactic category <Insert>. This method extracts three items: the table name, the field list, and the value list. The class *InsertData* holds these values and makes them available via accessor methods; see Figure 18-8.

```
public class InsertData {
   private String tblname;
   private List<String> flds;
   private List<Constant> vals;

   public InsertData(String tblname, List<String> flds,
                     List<Constant> vals) {
      this.tblname = tblname;
      this.flds = flds;
      this.vals = vals;
   }

   public String tableName() {
      return tblname;
   }

   public List<String> fields() {
      return flds;
   }

   public List<Constant> vals() {
      return vals;
   }
}
```

Figure 18-8: The code for the SimpleDB class *InsertData*

18.6.5 Parsing Deletions

Deletion statements are handled by the method *delete*. The method returns an object of class *DeleteData*; see Figure 18-9. The class constructor stores the table name and predicate from the specified deletion statement, and provides methods *tableName* and *pred* to access them.

```
public class DeleteData {
   private String tblname;
   private Predicate pred;

   public DeleteData(String tblname, Predicate pred) {
      this.tblname = tblname;
      this.pred = pred;
   }

   public String tableName() {
      return tblname;
   }

   public Predicate pred() {
      return pred;
   }
}
```

Figure 18-9: The code for the SimpleDB class *DeleteData*

18.6.6 Parsing Modifications

Modification statements are handled by the method *modify*. The method returns an object of class *ModifyData*; see Figure 18-10. This class is very similar to that of *DeleteData*. The difference is that this class also holds the assignment information: the fieldname of the left-hand-side of the assignment, and the expression of the right-hand-side of the assignment. The additional methods *targetField* and *newValue* return this information.

```
public class ModifyData {
   private String tblname;
   private String fldname;
   private Expression newval;
   private Predicate pred;

   public ModifyData(String tblname, String fldname,
                     Expression newval, Predicate pred) {
      this.tblname = tblname;
      this.fldname = fldname;
      this.newval = newval;
      this.pred = pred;
   }

   public String tableName() {
      return tblname;
   }

   public String targetField() {
      return fldname;
   }

   public Expression newValue() {
      return newval;
   }

   public Predicate pred() {
      return pred;
   }
}
```

Figure 18-10: The code for the SimpleDB class *ModifyData*

18.6.7 Parsing Table, View and Index Creation

The syntactic category <Create> specifies the three SQL creation statements supported by SimpleDB. Table creation statements are handled by the syntactic category <CreateTable> and its method *createTable*. The methods *fieldDef* and *fieldType* extract the information of one field, and save it in its own schema. The method *fieldDefs* then adds this schema to the table's schema. The table name and schema are returned inside a *CreateTableData* object; see Figure 18-11.

```
public class CreateTableData {
   private String tblname;
   private Schema sch;

   public CreateTableData(String tblname, Schema sch) {
      this.tblname = tblname;
      this.sch = sch;
   }

   public String tableName() {
      return tblname;
   }

   public Schema newSchema() {
      return sch;
   }
}
```

Figure 18-11: The code for the SimpleDB class *CreateTableData*

View creation statements are handled by the method *createView*. The method extracts the name and definition of the view, and returns them in an object of type *CreateViewData*; see Figure 18-12. The handling of the view definition is unusual. It needs to be parsed as a <Query>, in order to detect badly-formed view definitions. However, the metadata manager doesn't want to save the parsed representation of the definition; it wants the actual query string. Consequently, the *CreateViewData* constructor re-creates the view definition by calling *toString* on the returned *QueryData* object. In effect, the *toString* method "unparses" the query.

```
public class CreateViewData {
      private String viewname;
      private QueryData qrydata;

      public CreateViewData(String viewname, QueryData qrydata) {
         this.viewname = viewname;
         this.qrydata = qrydata;
      }

      public String viewName() {
         return viewname;
      }

      public String viewDef() {
         return qrydata.toString();
      }
}
```

Figure 18-12: The code for the SimpleDB class *CreateViewData*

An index is a data structure that can be used to improve query efficiency; Chapter 6 discussed the use of indexes, and Chapter 21 will discuss their implementation. The *createIndex* parser method extracts the index name, table name and field name and saves it in a *CreateIndexData* object; see Figure 18-13.

```
public class CreateIndexData {
   private String idxname, tblname, fldname;

   public CreateIndexData(String idxname, String tblname,
                          String fldname) {
      this.idxname = idxname;
      this.tblname = tblname;
      this.fldname = fldname;
   }

   public String indexName() {
      return idxname;
   }

   public String tableName() {
      return tblname;
   }

   public String fieldName() {
      return fldname;
   }
}
```

Figure 18-13: The code for the SimpleDB class *CreateIndexData*

18.7 Chapter Summary

- The *syntax* of a language is a set of rules that describe the strings that could possibly be meaningful statements.

- The *parser* is responsible for ensuring that its input string is syntactically correct.

- The *lexical analyzer* is the portion of the parser that splits the input string into a series of *tokens*.

- Each token has a *type* and a *value*. The SimpleDB lexical analyzer supports five token types:
  - single-character *delimiters*, such as the comma;
  - *integer constants*, such as *123*;
  - *string constants*, such as *'joe'*;
  - *keywords*, such as *select*, *from*, and *where*;
  - *identifiers*, such as *STUDENT*, *x*, and *glop34a*.

- Each token type has two methods: methods that ask about the current token, and methods that tell the lexical analyzer to “eat” the current token, returning its value and moving to the next token.

- A *grammar* is a set of rules that describe how tokens can be legally combined.
    - The left side of a grammar rule specifies its *syntactic category*. A syntactic category denotes a particular concept in the language.
    - The right side of a grammar rule specifies the *contents* of that category, which is the set of strings that satisfy the rule.

- A *parse tree* has syntactic categories as its internal nodes, and tokens as its leaf nodes. The children of a category node correspond to the application of a grammar rule. A string is in a syntactic category iff it has a parse tree having that category as its root.

- A *parsing algorithm* constructs a parse tree from a syntactically legal string. The complexity of the parsing algorithm is usually in proportion to the complexity of the grammars it can support. A simple parsing algorithm is known as *recursive descent*.

- A recursive-descent parser has a method for each grammar rule. Each method calls the methods corresponding to the items in the right side of the rule.

- Each method in a recursive-descent parser extracts the values of the tokens it reads, and returns them. An SQL parser should extract information such as table names, field names, predicates, and constants from the SQL statement. What gets extracted depends on the kind of SQL statement it is:
    - *For a query*: a collection of field names (from the *select* clause), a collection of table names (from the *from* clause), and a predicate (from the *where* clause).
    - *For an insertion*: a table name, a list of field names, and a list of values.
    - *For a deletion*: a table name and a predicate.
    - *For a modification*: a table name, the name of the field to be modified, an expression denoting the new field value, and a predicate.
    - *For a table creation*: a table name and its schema.
    - *For a view creation*: a table name and its definition.
    - *For index creation*: an index name, a table name, and the name of the indexed field.

18.8 Suggested Reading

The area of lexical analysis and parsing has received a tremendous amount of attention, going back over 50 years. The book [Scott 2000] gives an excellent introduction to the various algorithms in current use. Numerous SQL parsers are available over the web, such as Zql (*www.experlog.com/gibello/zql*). An SQL grammar can be found in the appendix of [Date and Darwen, 2004]. An online earlier version of the grammar is at the URL *http://www.contrib.andrew.cmu.edu/~shadow/sql/sql2bnf.aug92.txt*. (A copy of the SQL-92 standard, which describes SQL and its grammar, is at the URL *http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt*. If you have never looked at a standards document, you should check this out just for the experience.)

18.9 Exercises

CONCEPTUAL PROBLEMS

18.1 Draw a parse tree for the following SQL statements.

a) `select a from x where b = 3`

b) `select a, b from x,y,z`

c) `delete from x where a = b and c = 0`

d) `update x set a = b where c = 3`

e) `insert into x (a,b,c) values (3, 'glop', 4)`

f) `create table x ( a varchar(3), b int, c varchar(2))`

18.2 For each of the following strings, state where the exception will be generated when it is parsed, and why. Then execute each query from a JDBC client and see what happens.

a) `select from x`

b) `select x x from x`

c) `select x from y z`

d) `select a from where b=3`

e) `select a from y where b -=3`

f) `select a from y where`

18.3 The parser method *create* does not correspond to the SQL grammar of Figure 18-4.

a) Explain why the grammar rule for <Create> is too ambiguous to be used for recursive-descent parsing.

b) Revise the grammar so that it corresponds to how the *create* method actually works.

PROGRAMMING PROBLEMS

18.3 Revise each parser method corresponding to a recursive rule so that it uses a while-loop instead of recursion.

18.4 Revise the class *PredParser* (from Figure 18-5) to print out the parse tree resulting from the sequence of method calls.

18.5 Exercise 17.14 asked you to implement null values. In this exercise we revise SQL to understand nulls.

a) Revise the SimpleDB grammar to accept the keyword “null” as a constant.

b) Revise the SimpleDB parser to implement the grammar change from part (a).

c) In standard SQL, a term can be of the form “*GradYear is null*”, which returns true if the expression “*GradYear*” is a null value. The two keywords “*is null*” are treated as a single operator having one argument. Revise the SimpleDB grammar to have this new operator.

d) Revise the SimpleDB parser and the class *Term* to implement the grammar changes from part (c).

e) Write a program to in JDBC to test your code. Your program can set values to null (or use an unassigned value of a newly inserted record), and then execute a query that involves “is null”. Note that your program will not be able to print null values until you modify JDBC; see Exercise 20.6.

18.6 Exercise 17.11 asked you to modify expressions to handle arithmetic expressions.
a) Revise the SQL grammar similarly.
b) Revise the SimpleDB parser to implement the grammar changes.
c) Write a JDBC client to test the server. For example, write a program to increment the graduation year of all students having major #30.

18.7 Exercise 17.12 asked you to modify terms.
a) Revise the SQL grammar similarly.
b) Revise the SimpleDB parser to implement the grammar changes.
c) Write a JDBC client to test the server. For example, write a program that prints the names of all students who graduated before 1980.

18.8 Exercise 17.13 asked you to modify predicates.
a) Revise the SQL grammar similarly.
b) Revise the SimpleDB parser to implement the grammar changes.
c) Write a JDBC client to test the server. For example, write a program that prints the names of all students having majors 10 or 20.

18.9 SimpleDB also does not allow parentheses in its predicates.
a) Revise the SQL grammar appropriately (either with or without having done Exercise 18.8).
b) Revise the SimpleDB parser to implement the grammar changes.
c) Write a JDBC client to test the server.

18.10 Recall from Chapter 4 that join predicates can be specified in standard SQL by means of the JOIN keyword in the from clause.
a) Revise the SQL lexical analyzer to include the keywords “join” and “on”.
b) Revise the SQL grammar to handle explicit joins.
c) Revise the SimpleDB parser to implement your grammar changes. Add the join predicate to the predicate you get from the where clause.
d) Write a JDBC program that tests out your changes.

18.11 Recall from Chapter 4 that fields in standard SQL can be optionally qualified by a *range variable* that denotes the table it comes from. Range variables are (optionally) specified in the *from* clause of an SQL statement.
a) Revise the SimpleDB grammar to allow for this feature.
b) Revise the SimpleDB parser to implement your grammar changes. You will also have to modify the information returned by the parser, and possibly make minor changes to the planner. Note that you will not be able to test your changes on the SimpleDB server unless you also extend the planner; see Exercise 19.9.

18.12 Recall from Chapter 4 that the keyword AS can be used in standard SQL to extend the output table with computed values.
a) Revise the SimpleDB grammar to allow an optional AS expression after any fields in the select clause.
b) Revise the SimpleDB lexical analyzer and parser to implement your grammar changes. How should the parser make this additional information available? Note that you will not be able to test your changes on the SimpleDB server unless you also extend the planner; see Exercise 19.10.

18.13 Recall from Chapter 4 that the keyword UNION can be used in standard SQL to combine the output tables of two queries.
a) Revise the SimpleDB grammar to allow a query to be the union of two other queries.
b) Revise the SimpleDB lexical analyzer and parser to implement your grammar changes. Note that you will not be able to test your changes on the SimpleDB server unless you also extend the planner; see Exercise 19.11.

18.14 Recall from Chapter 4 that standard SQL supports nested queries.
a) Revise the SimpleDB grammar to allow a term to be of the form "*fieldname op query*", where *op* is either "in" or "not in".
b) Revise the SimpleDB lexical analyzer and parser to implement your grammar changes. Note that you will not be able to test your changes on the SimpleDB server unless you also extend the planner; see Exercise 19.12.

18.15 In standard SQL, the "*" character can be used in the select clause to denote all fields of a table. If SQL supports range variables (as in Exercise 18.11), then the "*" can likewise be prefixed by a range variable.
a) Revise the SimpleDB grammar to allow "*" to appear in queries.
b) Revise the SimpleDB parser to implement your grammar changes. Note that you will not be able to test your changes on the SimpleDB server unless you also extend the planner; see Exercise 19.13.

18.16 In Standard SQL one can insert records into a table via the following variant on the insert statement:

```
insert into MATHSTUDENT(SId, SName)
select STUDENT.SId, STUDENT.SName
from STUDENT, DEPT
where STUDENT.MajorId = DEPT.DId and DEPT.DName = 'math'
```

That is, the records returned by the select statement are inserted into the specified table. (The above statement assumes that the *MATHSTUDENT* table has already been created.)
a) Revise the SimpleDB SQL grammar to handle this form of insertion.
b) Revise the SimpleDB parser code to implement your grammar. Note that you will not be able to run JDBC queries until you also modify the planner; see Exercise 19.14.

18.17 Exercise 17.10 asked you to create new kinds of constant.

a) Modify the SimpleDB SQL grammar to allow these types to be used in a create table statement.
b) Do you need to introduce new constant literals? If so, modify the <Constant> syntactic category.
c) Revise the SimpleDB parser code to implement your grammar.

18.18 The open-source software package *javacc* (see the URL *javacc.dev.java.net*) builds parsers from grammar specifications. Use javacc to create a parser for the SimpleDB grammar. Then replace the existing parser with your new one.

18.19 The class *Parser* contains a method for each syntactic category in the grammar. Our simplified SQL grammar is small, and so the class is manageable. However, a full-featured grammar would cause the class to be significantly larger. An alternative implementation strategy is to put each syntactic category in its own class. The constructor of the class would perform the parsing for that category. The class would also have methods that returned the values extracted from the parsed tokens. This strategy creates a large number of classes, each of which is relatively small. Rewrite the SimpleDB parser using this strategy.

19
PLANNING
and the package simpledb.planner

The parser extracts the relevant data from an SQL statement. The next step is to turn that data into a relational algebra query tree. This step is called *planning*. In this chapter we examine the basic planning process. In particular, we will examine what the planner needs to do to verify that an SQL statement is semantically meaningful, look at some rudimentary plan-construction algorithms, and see how update plans get executed.

An SQL statement can have many equivalent query trees, often with wildly different costs. A good planning algorithm is essential for any database system that hopes to be commercially viable. The planning algorithms in most commercial systems are large, complex, and highly tuned, in the hope of finding the most efficient plans. In this chapter we totally ignore the efficiency issue, and focus exclusively on how planners work. We take up the topic of efficient planning algorithms in Chapter 24.

19.1 The SimpleDB Planner

> *The planner is the component of the database system*
> *that transforms an SQL statement into a plan.*

In order to study the planning process, we shall begin with the SimpleDB planner.

19.1.1 Using the planner

The SimpleDB planner is implemented by the class *Planner*, whose API consists of two methods; see Figure 19-1.

<u>*Planner*</u>

```
public Plan createQueryPlan(String query, Transaction tx);
public int  executeUpdate(String cmd, Transaction tx);
```

Figure 19-1: The API for the SimpleDB planner

The first argument of both methods is a string representation of an SQL statement. The method *createQueryPlan* creates and returns a plan for the input query string. The method *executeUpdate* executes its command string, and returns the number of affected records (the same as the *executeUpdate* method in JDBC).

A client can obtain a *Planner* object by calling the static method *planner* in the class *simpledb.server.SimpleDB*. Figure 19-2 contains some example code illustrating the use of the planner.

```
SimpleDB.init("studentdb");
Planner planner = SimpleDB.planner();
Transaction tx  = new Transaction();

// part 1: Process a query
String qry = "select sname, gradyear from student";
Plan p = planner.createQueryPlan(qry, tx);
Scan s = p.open();
while (s.next())
      System.out.println(s.getString("sname") + " " +
                         s.getInt("gradyear"));
s.close();

// part 2: Process an update command
String cmd = "delete from STUDENT where MajorId = 30";
int num = planner.executeUpdate(cmd, tx);
System.out.println(num + " students were deleted");

tx.commit();
```

Figure 19-2: Using the SimpleDB planner

Part 1 of the code shows how an SQL query is processed. The query string is passed into the planner's *createQueryPlan* method, and a plan is returned. Opening that plan returns a scan, whose records are then accessed and printed. Part 2 of the code shows how an SQL update command is processed. The command string is passed into the planner's *executeUpdate* method, which performs all of the necessary work.

19.1.2 What does the planner do?

The SimpleDB planner has two methods: one to handle queries, and one to handle updates. These methods both process their input quite similarly; Figure 19-3 lists the steps they take.

1. *Parse the SQL statement*. The method calls the parser, passing it the input string; the parser will return an object containing the data from the SQL statement. For example, the parser will return a *QueryData* object for a query, an *InsertData* object for an insert statement, and so on.
2. *Verify the SQL statement*. The method examines the *QueryData* (or *InsertData*, etc.) object to determine if it is semantically meaningful.
3. *Create a plan for the SQL statement*. The method uses a planning algorithm to determine a query tree corresponding to the statement, and to create a plan corresponding to that tree.

4a. *Return the plan* (for the *createQueryPlan* method).

4b. *Execute the plan* (for the *executeUpdate* method). The method creates a scan by opening the plan; then it iterates through the scan, making the appropriate update for each record in the scan and returning the number of records affected.

Figure 19-3: The steps taken by the two planner methods

In particular, both methods perform steps 1-3. The methods differ primarily in what they do with the plan that they create. The method *createQueryPlan* simply returns its plan, whereas *executeUpdate* opens and executes its plan.

Step 1 involves parsing, which was described in the previous chapter. This chapter describes the remaining three steps.

19.2 Verification

The first responsibility of the planner is to determine whether its SQL statement is actually meaningful.

The planner needs to verify the following things about an SQL statement:
- *The mentioned tables and fields actually exist in the catalog.*
- *The mentioned fields are not ambiguous.*
- *The actions on fields are type-correct.*
- *All constants are the correct size and type for their fields.*

All of the information required to perform this verification can be found by examining the schemas of the mentioned tables. For example, the absence of a schema indicates that the mentioned table does not exist; the absence of a field in any of the schemas indicates that the field does not exist, and its presence in multiple schemas indicates the possibility of ambiguity. In addition, the planner can use the type and length of each mentioned field to determine type-correctness.

Type-correctness needs to be verified in predicates, modification assignments, and insertions. For a predicate, the arguments to each operator in an expression must be of compatible types, as must the expressions in each term. A modification assigns an expression to a field; both of these types must be compatible. And for an insertion statement, the type of each inserted constant must be compatible with the type of its associated field.

The SimpleDB planner can obtain the table schemas from the metadata manager, via the *getTableInfo* method. However, it currently does not perform any explicit verification. Exercises 19.4-19.8 ask you to rectify this situation.

19.3 Query Planning

We now tackle the problem of how to construct a plan out of the data returned by the parser. This section considers the query-planning problem. The next section considers planning for update statements.

19.3.1 The SimpleDB query planning algorithm

SimpleDB supports a simplified subset of SQL that contains no computation, no sorting, no grouping, no nesting, and no renaming. Consequently, all of its SQL queries can be implemented by a query tree that uses only the three operators *select*, *project*, and *product*. An algorithm for creating such a plan appears in Figure 19-4.

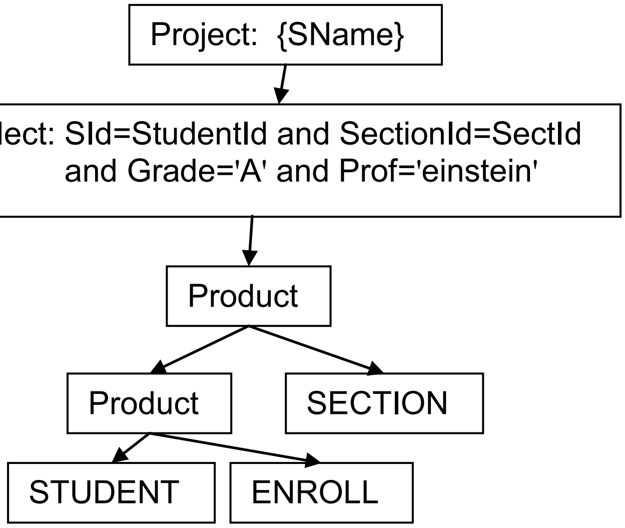
1. Construct a plan for each table T in the *from* clause.
   a) If T is a stored table, then the plan is a table plan for T.
   b) If T is a view, then the plan is the result of calling this algorithm recursively on the definition of T.
2. Take the product of the tables in the *from* clause, in the order given.
3. Select on the predicate in the *where* clause.
4. Project on the fields in the *select* clause.

Figure 19-4: The basic query planning algorithm for the SimpleDB subset of SQL

Figures 19-5 and 19-6 give two examples of this query planning algorithm. Both figures contain queries that retrieve the name of those students who received an ‘A’ with professor Einstein.

```
select SName
from STUDENT, ENROLL, SECTION
where SId = StudentId
and SectionId = SectId
and Grade = 'A'
and Prof = 'einstein'
```

(a)


Project: {SName}
Select: SId=StudentId and SectionId=SectId and Grade='A' and Prof='einstein'
Product
Product
SECTION
STUDENT
ENROLL

(b)

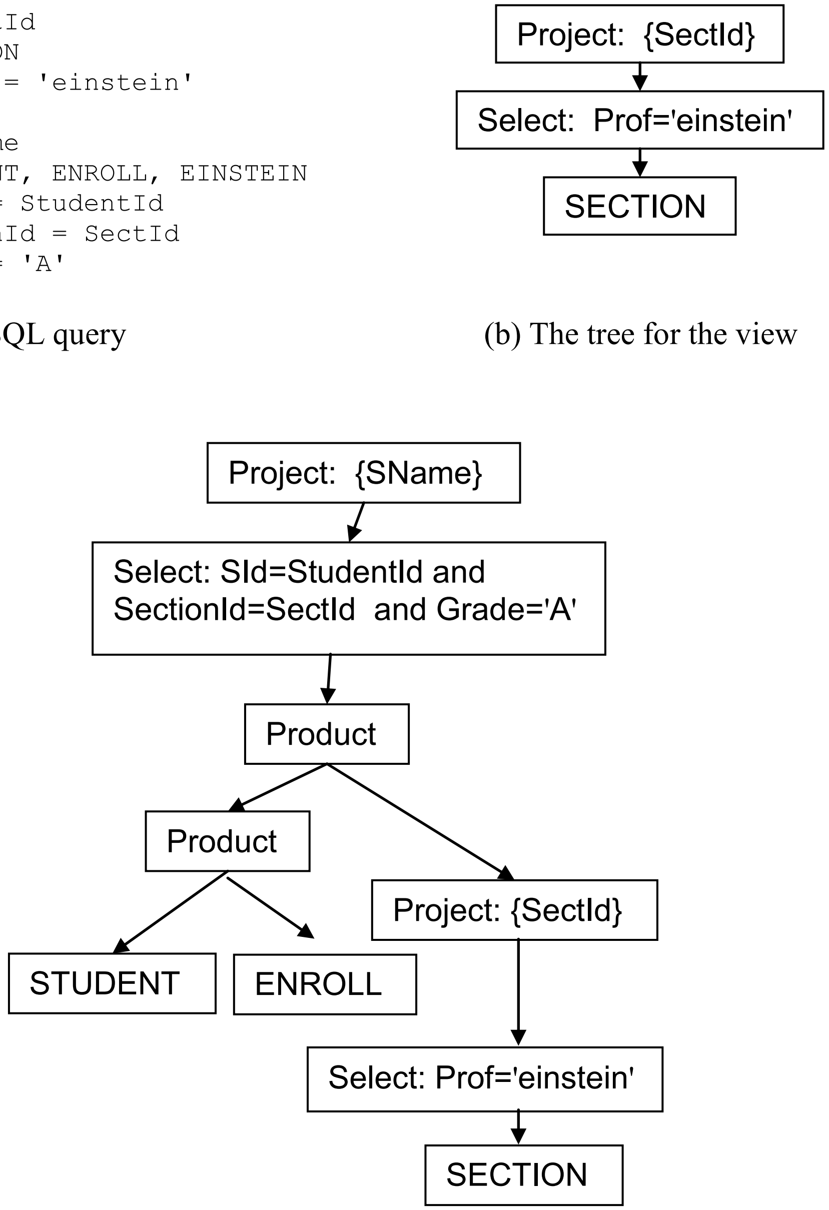
Figure 19-5: Applying the basic query planning algorithm to an SQL query

The query in Figure 19-5(a) does not use views. Its corresponding query tree appears in Figure 19-5(b). Note how the tree consists of the product of all three tables, followed by a selection and a projection.

The query in Figure 19-6(a) uses a view. Figure 19-6(b) depicts the tree that results from calling the algorithm on the view's definition, and Figure 19-6(c) depicts the tree for the entire query. Note how the final tree consists of the product of the two tables and the view tree, followed by a selection and a projection. Also note that this final tree is equivalent to, but somewhat different from, the tree of Figure 19-5(b). In particular, part of the original selection predicate has been "pushed" down the tree, and there is an intermediate projection. Equivalences such as this are important to query optimization, as we shall see in Chapter 24.

```
create view EINSTEIN as
select SectId
from SECTION
where Prof = 'einstein'

select SName
from STUDENT, ENROLL, EINSTEIN
where SId = StudentId
and SectionId = SectId
and Grade = 'A'
```

(a) The SQL query



(b) The tree for the view

(c) The tree for the entire query

Figure 19-6: Applying the basic query planning algorithm in the presence of views

The basic query planning algorithm is rigid an naïve. It generates the product plans in the order returned by the method *QueryData.tables*. Note that this order is completely arbitrary – any other ordering of the tables will produce an equivalent scan. Unfortunately, equivalent scans may require significantly different numbers of block accesses. The performance of this algorithm will therefore be erratic (and often poor), because it creates an arbitrary plan. The planning algorithms in commercial database systems are much more sophisticated than the algorithm of Figure 19-4. They not only analyze the cost of many equivalent plans, they also implement additional relational operations that can be applied in special circumstances. Their goal is to choose the most efficient plan (and thereby be more desirable than their competition). These techniques are the subject of Part 4 of this book.

The SimpleDB class *BasicQueryPlanner* implements the basic query planning algorithm; its code appears in Figure 19-7. Each of the four steps in the code implements the corresponding step in that algorithm.

```
public class BasicQueryPlanner implements QueryPlanner {

   public Plan createPlan(QueryData data, Transaction tx) {

   //Step 1: Create a plan for each mentioned table or view
      List<Plan> plans = new ArrayList<Plan>();
      for (String tblname : data.tables()) {
         String viewdef = SimpleDB.mdMgr().getViewDef(tblname, tx);
         if (viewdef != null)
            plans.add(SimpleDB.planner().createQueryPlan(viewdef, tx));
         else
            plans.add(new TablePlan(tblname, tx));
      }

   //Step 2: Create the product of all plans for mentioned tables
      Plan p = plans.remove(0);
      for (Plan nextplan : plans)
         p = new ProductPlan(p, nextplan);

   //Step 3: Add a selection plan for the predicate
      p = new SelectPlan(p, data.pred());

   //Step 4: Project on the field names
      p = new ProjectPlan(p, data.fields());
      return p;
   }
}
```

Figure 19-7: The code for the SimpleDB class *BasicQueryPlanner*

19.3.2 Planning for Standard SQL Queries

A SimpleDB SQL query can be translated into a query tree whose nodes come from the three relational algebra operators *product*, *select*, and *project*. A query in standard SQL has additional features that require other relational algebra operators:

- Nested queries use the *semijoin* and *antijoin* operators.
- Aggregations use the *groupby* operator.
- Renamed and computed values use the *extend* operator.
- Outer joins use the *outerjoin* operator.
- Unions use the *union* operator.
- Sorting uses the *sort* operator.

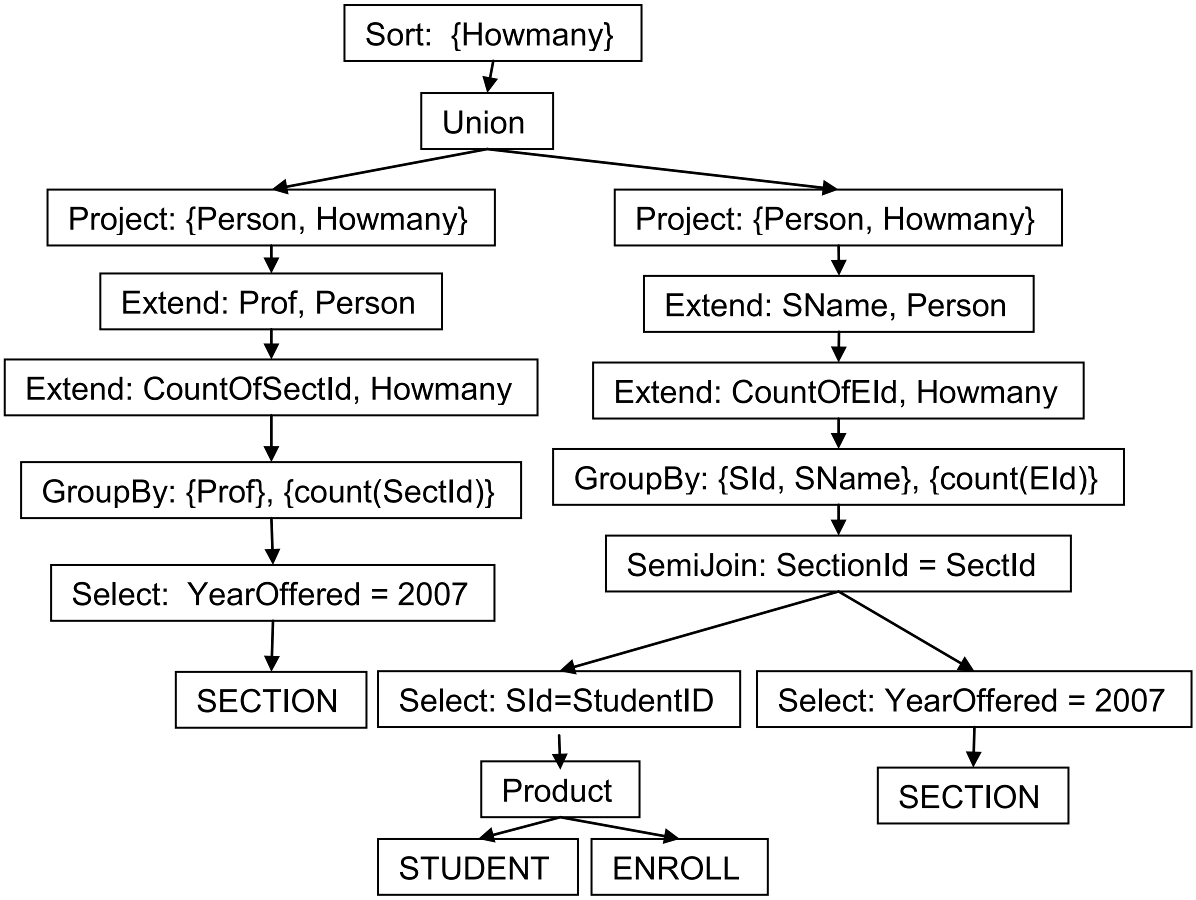
The semantics of an SQL query requires that the operators appear in a particular order in the query tree (not considering the subtrees created by views). For SimpleDB queries, we saw that the *product* operators are at the bottom of the query tree, followed by a single *select* operator and then a *project* operator. In standard SQL the ordering is as follows, starting from the bottom:

the *product* operators;
the *outerjoin* operators;
the *select*, *semijoin* and *antijoin* operators;
the *extend* operators;
the *project* operator;
the *union* operators;
the *sort* operator;

For example, consider the SQL query of Figure 19-8(a), which returns information about people who either took or taught sections in 2007. The first part of the union calculates the number of sections taken by each student in 2007; the second part of the union calculates the number of sections taught by each professor in 2007. The union is sorted by the number of sections taken (or taught). The query tree of Figure 19-8(b) depicts the straightforward translation of this query into relational algebra, without any consideration of efficiency.

```
select SName as Person, count(EId) as Howmany
from STUDENT, ENROLL
where SId = StudentId
and SectionId in (select SectId
                  from SECTION
                  where YearOffered = 2007)
group by SId, SName
union
select Prof as Person, count(SectId) as Howmany
from SECTION
where YearOffered = 2007
group by Prof
order by Howmany
```

(a) A complex SQL query


Sort: {Howmany}
Union
Project: {Person, Howmany}
Project: {Person, Howmany}
Extend: Prof, Person
Extend: SName, Person
Extend: CountOfSectId, Howmany
Extend: CountOfEId, Howmany
GroupBy: {Prof}, {count(SectId)}
GroupBy: {SId, SName}, {count(EId)}
SemiJoin: SectionId = SectId
Select: YearOffered = 2007
SECTION
Select: SId=StudentID
Select: YearOffered = 2007
Product
SECTION
STUDENT
ENROLL

(b) The query tree produced by the basic planning algorithm

Figure 19-8: Extending the basic query planner to include other operators

In order for SimpleDB to be able to generate such a query tree, its parser needs to be extended so that it extracts the necessary additional information from the query. Some of this additional information comes from new clauses in the SQL query, such as the *order by* and *group by* clauses. In this case, the grammar rule for <Query> will need to be modified to include the corresponding additional syntactic categories. Some of the additional information comes from enhancements to existing clauses, such as the computations performed in the *select* clause and the explicit joins in the *from* clause. In this case, the existing syntactic categories will need to be modified correspondingly.

The SimpleDB planner will also need to be extended to use the new information extracted by the parser. The planning algorithm builds a query tree bottom-up, by first generating the product nodes, then the outerjoin nodes, etc. This task is easy in Figure 19-7, because the information for each operator appears in a separate clause. In the general case, however, the planner will need to search through the parser information to find what it needs. For example, the *select* clause can contain information applicable to the *extend*, *groupby*, and *project* operators. The planner must examine this information to determine which portion of it applies to which operator. For another example, the parser information about a nested query will appear as a term in the query's predicate. The planner will need to extract these terms when building the *select*, *semijoin*, and *antijoin* nodes.

19.4 Update Planning

We now consider how a planner should process update statements. The SimpleDB class *BasicUpdatePlanner* provides the most straightforward implementation of an update planner; see Figure 19-9. This class contains one method for each kind of update. These methods are discussed in the following subsections.

```
public class BasicUpdatePlanner implements UpdatePlanner {

   public int executeDelete(DeleteData data, Transaction tx) {
      Plan p = new TablePlan(data.tableName(), tx);
      p = new SelectPlan(p, data.pred());
      UpdateScan us = (UpdateScan) p.open();
      int count = 0;
      while(us.next()) {
         us.delete();
         count++;
      }
      us.close();
      return count;
   }

   public int executeModify(ModifyData data, Transaction tx) {
      Plan p = new TablePlan(data.tableName(), tx);
      p = new SelectPlan(p, data.pred());
      UpdateScan us = (UpdateScan) p.open();
      int count = 0;
      while(us.next()) {
         Constant val = data.newValue().evaluate(us);
         us.setVal(data.targetField(), val);
         count++;
      }
      us.close();
      return count;
   }

   public int executeInsert(InsertData data, Transaction tx) {
      Plan p = new TablePlan(data.tableName(), tx);
      UpdateScan us = (UpdateScan) p.open();
      us.insert();
      Iterator<Constant> iter = data.vals().iterator();
      for (String fldname : data.fields()) {
         Constant val = iter.next();
         us.setVal(fldname, val);
      }
      us.close();
      return 1;
   }

   public int executeCreateTable(CreateTableData data, Transaction tx){
      SimpleDB.mdMgr().createTable(data.tableName(),
                                   data.newSchema(), tx);
      return 0;
   }

   public int executeCreateView(CreateViewData data, Transaction tx) {
      SimpleDB.mdMgr().createView(data.viewName(), data.viewDef(), tx);
      return 0;
   }

   public int executeCreateIndex(CreateIndexData data, Transaction tx){
      SimpleDB.mdMgr().createIndex(data.indexName(), data.tableName(),
                                   data.fieldName(), tx);
      return 0;
   }
}
```

Figure 19-9: The code for the SimpleDB class *BasicUpdatePlanner*

19.4.1 Delete and modify planning

The scan for a delete (or modify) statement is a select scan that retrieves those records to be deleted (or modified). For example, consider the following modification statement:

```
update STUDENT
set MajorId = 20
where MajorId = 30 and GradYear = 2008
```

and the following deletion statement:

```
delete from STUDENT
where MajorId = 30 and GradYear = 2008
```

These statements have the same scan, namely all students in department 30 graduating in 2008. The methods *executeDelete* and *executeModify* create and iterate through this scan, performing the appropriate action on each of its records. In the case of the modification statement, each record is modified; in the case of the deletion statement, each record is deleted.

Looking at the code, we see that both methods create the same plan, which is similar to the plan created by the query planner (except that the query planner would also add a project plan). Both methods also open the scan and iterate through it in the same way. The *executeDelete* method calls *delete* on each record in the scan, whereas *executeModify* performs a *setVal* operation on the modified field of each record in the scan. Both methods also keep a count of the affected records, which is returned to the caller.

19.4.2 Insert planning

The scan corresponding to an insert statement is simply a table scan of the underlying table. The *executeInsert* method begins by inserting a new record into this scan. It then iterates through the *fields* and *vals* lists in parallel, calling *setInt* or *setString* to modify the value of each specified field of the record. The method returns a 1, denoting that 1 record was inserted.

19.4.3 Planning for table, view, and index creation

The code for the methods *executeCreateTable*, *executeCreateView*, and *executeCreateIndex* are different from the others, because they don't require accessing any data records, and thus don't require a scan. They simply call the metadata methods *createTable, createView*, and *createIndex*, using the appropriate information from the parser; they return 0 to indicate that no records were affected.

19.5 Implementing the SimpleDB Planner

Recall from Figure 19-1 that the SimpleDB planner is implemented by the class *Planner*, which contains the methods *createQueryPlan* and *executeUpdate*. We are now ready to examine how these methods are implemented; see Figure 19-10.

```
public class Planner {
   private QueryPlanner  qplanner;
   private UpdatePlanner uplanner;

   public Planner(QueryPlanner qplanner, UpdatePlanner uplanner) {
      this.qplanner = qplanner;
      this.uplanner = uplanner;
   }

   public Plan createQueryPlan(String cmd, Transaction tx) {
      Parser parser = new Parser(cmd);
      QueryData data = parser.query();
      // code to verify the query should be here...
      return qplanner.createPlan(data, tx);
   }

   public int executeUpdate(String cmd, Transaction tx) {
      Parser parser = new Parser(cmd);
      Object obj = parser.updateCmd();
      // code to verify the update command should be here ...

      if (obj instanceof InsertData)
         return uplanner.executeInsert((InsertData)obj, tx);
      else if (obj instanceof DeleteData)
         return uplanner.executeDelete((DeleteData)obj, tx);
      else if (obj instanceof ModifyData)
         return uplanner.executeModify((ModifyData)obj, tx);
      else if (obj instanceof CreateTableData)
         return uplanner.executeCreateTable((CreateTableData)obj, tx);
      else if (obj instanceof CreateViewData)
         return uplanner.executeCreateView((CreateViewData)obj, tx);
      else if (obj instanceof CreateIndexData)
         return uplanner.executeCreateIndex((CreateIndexData)obj, tx);
      else
         return 0;  // this option should never occur
   }
}
```

Figure 19-10: The code for the SimpleDB class *Planner*

The methods are a straightforward implementation of Figure 19-3. The method *createQueryPlan* creates a parser for its input SQL query, calls the parser method *query* to parse the string, verifies the returned *QueryData* object (at least, the method ought to), and returns the plan generated by the query planner. The method *executeUpdate* is similar: It parses the update string, verifies the object returned by the parser, and calls the appropriate update planner method to perform the execution.

Note that the object returned by the update parser will be of type *InsertData*, *DeleteData*, etc., according to what kind of update statement was submitted. The *executeUpdate* code checks this type in order to determine which planner method to call.

The *Planner* object depends on its query planner and update planner to do the actual planning. These objects are passed into the *Planner* constructor, which allows us to configure the planner with different planning algorithms. For example, Chapter 24 develops a fancy query planner called *HeuristicQueryPlanner*; we can use this planner instead of *BasicQueryPlanner* if we want, simply by passing a *HeuristicQueryPlanner* object into the *Planner* constructor.

Java interfaces are used to obtain this plug-and-play capability. The arguments to the *Planner* constructor belong to the interfaces *QueryPlanner* and *UpdatePlanner*, whose code appears in Figure 19-11.

```
public interface QueryPlanner {
   public Plan createPlan(QueryData data, Transaction tx);
}

public interface UpdatePlanner {
   public int executeInsert(InsertData data, Transaction tx);
   public int executeDelete(DeleteData data, Transaction tx);
   public int executeModify(ModifyData data, Transaction tx);
   public int executeCreateTable(CreateTableData data, Transaction tx);
   public int executeCreateView(CreateViewData data, Transaction tx);
   public int executeCreateIndex(CreateIndexData data, Transaction tx);
}
```

Figure 19-11: The code for the SimpleDB *QueryPlanner* and *UpdatePlanner* interfaces

The *BasicQueryPlanner* and *BasicUpdatePlanner* classes implement these interfaces, as must every query and update planner we create.

Planner objects are created in the method *SimpleDB.planner*, which is in the package *simpledb.server*. That code creates a new basic query planner and a new basic update planner and passes them to the *Planner* constructor, as shown in Figure 19-12. To reconfigure SimpleDB to use a different query planner, you just need to modify the code so that it creates a different object instead of a *BasicQueryPlanner* object.

```
   public static Planner planner() {
      QueryPlanner  qplanner = new BasicQueryPlanner();
      UpdatePlanner uplanner = new BasicUpdatePlanner();
      return new Planner(qplanner, uplanner);
   }
```

Figure 19-12: The code for the method *SimpleDB.planner*

19.6 Chapter Summary

- The *planner* is the component of the database system that transforms an SQL statement into a plan.
- The planner *verifies* the statement is semantically meaningful, by checking that:

  - the mentioned tables and fields actually exist in the catalog;
  - the mentioned fields are not ambiguous;
  - the actions on fields are type-correct;
  - all constants are the correct size and type for their fields.

- The *basic query planning algorithm* creates a rudimentary plan, as follows:
  1. Construct a plan for each table T in the *from* clause.
     a) If T is a stored table, then the plan is a tableplan for T.
     b) If T is a view, then the plan is the result of calling this algorithm recursively on the definition of T.
  2. Take the product of the tables in the *from* clause, in the order given;
  3. Select on the predicate in the *where* clause;
  4. Project on the fields in the *select* clause.

- The basic query planning algorithm generates a naïve and often inefficient plan. The planning algorithms in commercial database systems perform extensive analysis of the various equivalent plans, which will be described in Chapter 24.

- The basic query planning algorithm only applies to the subset of SQL supported by SimpleDB. The basic planning algorithm can be extended to standard SQL queries by generating operator in the following order:
  - the *product* operators
  - the *outerjoin* operators
  - the *select*, *semijoin* and *antijoin* operators
  - the *extend* operators
  - the *project* operator
  - the *union* operators
  - the *sort* operator

Each kind of update command gets planned differently.

- Delete and modify statements are treated similarly. The planner creates a select plan that retrieves those records to be deleted (or modified). The methods *executeDelete* and *executeModify* open the plan and iterate through the resulting scan, performing the appropriate action on each of its records. In the case of the modify statement, each record is modified; in the case of the delete statement, each record is deleted.

- The plan for an insert statement is a table plan for the underlying table. The *executeInsert* method opens the plan and inserts a new record into that resulting scan.

- The plans for the creation statements do not need to create plans, because they do not access any data. Instead, the methods call the appropriate metadata method to perform the creation.

19.7 Suggested Reading

The planner in this chapter understands only a small subset of SQL, and we have touched only briefly on planning issues for the more complex constructs. The article [Kim 1982] describes the problems with nested queries and proposes some solutions. The article [Chaudhuri 1998] discusses strategies for the more difficult aspects of SQL, including outer joins and nested queries.

19.8 Exercises

CONECPTUAL EXERCISES

19.1 For each of the following SQL statements, draw a picture of the plan that would be generated by the basic planner of this chapter.

a)

```
select SName, Grade
from STUDENT, COURSE, ENROLL, SECTION
where SId = StudentId and SectId = SectionId and CourseId = CId
and Title = 'Calculus'
```

b)

```
select SName
from STUDENT
where MajorId = 10 and SId in (select StudentId
                               from ENROLL
                               where Grade = 'C')
order by SName
```

c)

```
select Title, count(SId) as HowMany
from STUDENT, COURSE, ENROLL, SECTION
where SId = StudentId and SectId = SectionId and CourseId = CId
group by Title
```

19.2 For each of the queries in Exercise 19.1, explain what things the planner must check to verify its correctness.

19.3 For each of the following update statements, explain what things the planner must check to verify its correctness.

a)

```
insert into STUDENT(SId, SName, GradYear, MajorId)
values(120, 'abigail', 2012, 30)
```

b)

```
delete from STUDENT
where MajorId = 10 and SID in (select StudentId
                               from ENROLL
                               where Grade = 'F')
```

c)

```
update STUDENT
set GradYear = GradYear + 3
where MajorId in (select DId from DEPT where DName = 'drama')
```

PROGRAMMING EXERCISES

19.4 The SimpleDB planner does not verify that table names are meaningful.
a) What problem will occur when a non-existent table is mentioned in a query?
b) Fix the *Planner* class to verify table names. Throw a *BadSyntaxException* if the table is nonexistent.

19.5 The SimpleDB planner does not verify that field names exist and are unique.
a) What problem will occur when a non-existent field name is mentioned in a query?
b) What problem will occur when tables having common field names are mentioned in a query?
c) Fix the code to perform the appropriate verification.

19.6 The SimpleDB planner does not type-check predicates.
a) What problem will occur if a predicate in an SQL statement is not type-correct?
b) Fix the code to perform the appropriate verification.

19.7 The SimpleDB update planner doesn't check that string constants are the right size and type for the specified fields in an insert statement, nor does it verify that the size of the constant and field lists are the same. Fix the code appropriately.

19.8 The SimpleDB update planner doesn't verify that the assignment of a new value to the specified field in a modify statement is type-correct. Fix the code appropriately.

19.9 Exercise 18.11 asked you to modify the parser to allow range variables, and Exercise 17.15(b) asked you to implement the *rename* operator. Recall from Chapter 4 that range variables can be implemented by using renaming – First the planner renames each table field; then it adds the product, select, and project operators; and then it renames the fields back to their non-prefixed names.
a) Revise the basic query planner to perform this renaming.
b) Write a JDBC program to test out your code. In particular, write a JDBC program that performs a self join, such as finding the students having the same major as Joe.

19.10 Exercise 18.12 asked you to modify the parser to allow the AS keyword in the select clause, and Exercise 17.15(c) asked you to implement the *extend* operator.
a) Revise the basic query planner to add *extend* into the query plan.
b) Write a JDBC program to test out your code.

19.11 Exercise 18.13 asked you to modify the parser to allow the UNION keyword, and Exercise 17.15(d) asked you to implement the *union* operator.
a) Revise the basic query planner to add *union* into the query plan.
b) Write a JDBC program to test out your code.

19.12 Exercise 18.14 asked you to modify the parser to allow nested queries, and Exercises 17.15(e)(f) asked you to implement the *semijoin* and *antijoin* operators.
a) Revise the basic query planner to add these operators into the query plan.

b) Write a JDBC program to test out your code.

19.13 Exercise 18.15 asked you to modify the parser to allow "*" to appear in the select clause of a query.
a) Revise the planner appropriately.
b) Write a JDBC client to test out your code.

19.14 Exercise 18.16 asked you to modify the SimpleDB parser to handle a new kind of insert statement.
a) Revise the planner appropriately.
b) Write a JDBC client to test out your code.

19.15 Exercise 18.5 asked you to modify the SimpleDB parser to handle the "is null" operator, and Exercise 17.14 asked you to implement null-valued constants. Revise the *executeInsert* method of the basic update planner so that a null value is used as the value of each unmentioned field.

19.16 The basic update planner inserts its new record starting from the beginning of the table.
a) Design and implement a modification to the planner that efficiently inserts from the end of the table, or perhaps from the end of the previous insertion.
b) Compare the benefits of the two strategies. Which do you prefer?

19.17 The SimpleDB basic update planner assumes that the table mentioned in an update command is a stored table. As discussed in Chapter 4, standard SQL also allows the table to be the name of a view, provided that the view is updatable.
a) Revise the update planner so that views can be updated. The planner doesn't need to check for non-updatable views. It should just try to perform the update and throw an exception if something goes wrong. Note that you will need to modify *ProjectScan* to implement *UpdateScan* interface, as in Exercise 17.16.
b) Explain what the planner would have to do in order to verify that the view definition was updatable.

19.18 The SimpleDB basic update planner deals with view definitions by "unparsing" the query and saving the query string in the catalog. The basic query planner then has to reparse the view definition each time it is used in a query. An alternative approach is for the create-view planner to save the parsed version of the query data in the catalog, which can then be retrieved by the query planner.
a) Implement this strategy. (HINT: Use object-serialization in Java. Serialize the *QueryData* object, and use a *StringWriter* to encode the object as a string. The metadata method *getViewDef* can then reverse the process, reconstructing the *QueryData* object from the stored string.)
b) How does this implementation compare to the approach taken in SimpleDB?

20
THE DATABASE SERVER
and the package simpledb.remote

In this chapter, we examine the two ways that a client program can access a database system: the database system can be embedded in the client, or it can run as a server that accepts requests from the client. The embedded-database strategy requires no new code, but has limited applicability. On the other hand, the server-based strategy allows multiple users to share the database concurrently, but it requires additional code to implement the server and handle the JDBC requests. We shall see how Java RMI can be used to build a database server and to implement a JDBC driver for it.

20.1 Server Databases vs. Embedded Databases

There are two ways that an application program can access a database:

- It can access the database indirectly, via a server.
- It can access the database directly.

Part 2 of this book focused on the first approach. The idea is that the application program first calls a driver to establish a connection with the database server. It then accesses the database by sending one or more SQL statements to the server, which executes each statement on the program's behalf and returns the results.

In the second approach, the application program calls database methods directly, without the need for an intermediary server. This approach is possible only if the database system exposes an API to its clients. In this case, we say that the program *embeds* the database system.

> *An application program embeds the database system*
> *if it calls the system's methods directly.*

An example of such a program appears in Figure 20-1(a). That code creates a new transaction, calls the planner to get a plan for an SQL query, opens the plan to get a scan, iterates through the scan, and commits the transaction. Compare this code with the equivalent JDBC code of Figure 20-1(b).

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();

Planner planner = SimpleDB.planner();
String qry = "select sname, gradyear from student";
Plan p = planner.createQueryPlan(qry, tx);
Scan s = p.open();

while (s.next())
      System.out.println(s.getString("sname") + " " +
                         s.getInt("gradyear"));
s.close();
tx.commit();
```

(a) using an embedded database system

```
Driver d = new SimpleDriver();
String url = "jdbc:simpledb://localhost";
Connection conn = d.connect(url, null);

Statement stmt = conn.createStatement();
String qry = "select sname, gradyear from student";
ResultSet rs = stmt.executeQuery(qry);

while (rs.next())
      System.out.println(s.getString("sname") + " " +
                         s.getInt("gradyear"));
rs.close();
conn.commit();
```

(b) using JDBC code

Figure 20-1: Two equivalent application program fragments

The code in Figure 20-1(a) uses five classes from SimpleDB: *SimpleDB*, *Transaction*, *Planner*, *Plan*, and *Scan*. The JDBC code uses the interfaces *Driver*, *Connection*, *Statement*, and *ResultSet*. There is a close correspondence between these constructs, which is shown in the table of Figure 20-2. This table also lists the correspondence between *ResultSetMetaData* and *Schema*, which did not appear in Figure 20-1.

JDBC Interface	**SimpleDB Class**
Driver	SimpleDB
Connection	Transaction
Statement	Planner, Plan
ResultSet	Scan
ResultSetMetaData	Schema

Figure 20-2: The correspondence between JDBC interfaces and SimpleDB classes

The classes in each row of Figure 20-2 correspond to each other because they share a common purpose. For example, both *Connection* and *Transaction* manage the current transaction, the classes *Statement*, *Planner* and *Plan* process SQL statements, and *ResultSet* and *Scan* both iterate through the result of a query. By taking advantage of this correspondence, any JDBC program supported by SimpleDB can be translated into an equivalent one that embeds the database.

A program that uses an embedded database system does not (and cannot) share its database with other programs. An embedded database system is private. There are no other users, and no concurrent threads (unless the program creates the threads itself, as in Figure 14-30). In fact, if two programs share the same embedded database, then there is nothing to stop the database from becoming corrupted – The programs cannot control each other's modifications, because each program manages its own lock table.

The "embedded database system" strategy therefore makes sense only for certain applications – namely, ones that:

- cannot easily connect to a database server, or
- are the exclusive users of the data (i.e. they "own" the data).

An example of the first kind of application is an automobile GPS navigation system. A GPS application accesses a database of road maps and business locations. Since connecting to a database server from a moving automobile is slow and uncertain, an embedded database system can allow the application to operate smoothly and reliably.

An example of the second kind of application is a sensor monitoring system, as might be found in a nuclear reactor. Such a system receives periodic readings from various sensors, which it saves in its database and analyzes for indications of potentially dangerous conditions. This sensor data is used exclusively by the monitoring system, so there is no need to get a server involved.

One consequence of using an embedded database system is that the application must run the database code locally, which may be on a considerably slower machine than a database server. On the other hand, embeddable database systems can be customizable, allowing users to turn off unneeded features. For example, consider again a GPS navigation system. This system is read-only and does not run concurrent threads; thus it does not need recovery management, concurrency control, or the update planner. By removing these features from its embedded database, the GPS system becomes slimmer and faster – possibly even faster than if it had used a full-featured, shared database server on a high-end machine.

20.2 Client-Server Communication

We now turn to the question of how to implement a database server. There are two issues:

- How to implement the client-server communication.

- How to implement the JDBC interfaces.

This section addresses the first issue; the second issue is covered in Section 20.3.

20.2.1 Remote Method Invocation

SimpleDB implements client-server communication via the Java *Remote Method Invocation* (*RMI*) mechanism. In order to use RMI, a system defines one or more interfaces that extend the Java interface *Remote;* these are called *remote interfaces*. Each remote interface always has two implementing classes: a *stub* class, whose objects live on the client, and a *remote implementation* class, whose objects live on the server. When the client calls a method from a stub object, the method call is sent across the network to the server and executed there by the corresponding remote implementation object; the result is then sent back to the stub object on the client. In short, a remote method is called by the client (using the stub object) but executed on the server (using the remote implementation object).

SimpleDB defines five remote interfaces: *RemoteDriver*, *RemoteConnection*, *RemoteStatement*, *RemoteResultSet,* and *RemoteMetaData*; their code appears in Figure 20-3. These remote interfaces mirror their corresponding JDBC interfaces, with two differences:

- They only implement the basic JDBC methods shown in Figure 7-1.
- They throw a *RemoteException* (as required by RMI) instead of an *SQLException* (as required by JDBC).

```
public interface RemoteDriver extends Remote {
   public RemoteConnection connect() throws RemoteException;
}

public interface RemoteConnection extends Remote {
   public RemoteStatement createStatement() throws RemoteException;
   public void            close()           throws RemoteException;
}

public interface RemoteStatement extends Remote {
   public RemoteResultSet executeQuery(String qry) throws RemoteException;
   public int             executeUpdate(String cmd) throws RemoteException;
}

public interface RemoteResultSet extends Remote {
   public boolean        next()                      throws RemoteException;
   public int            getInt(String fldname)      throws RemoteException;
   public String         getString(String fldname)   throws RemoteException;
   public RemoteMetaData getMetaData()               throws RemoteException;
   public void           close()                     throws RemoteException;
}

public interface RemoteMetaData extends Remote {
   public int    getColumnCount()              throws RemoteException;
   public String getColumnName(int column)     throws RemoteException;
   public int    getColumnType(int column)     throws RemoteException;
   public int getColumnDisplaySize(int column) throws RemoteException;
}
```

Figure 20-3: The SimpleDB remote interfaces

To get a feel for how RMI works, consider the client-side code fragment of Figure 20-4.

```
RemoteDriver rdvr = ...
RemoteConnection rconn = rdvr.connect();
RemoteStatement  rstmt = rconn.createStatement();
```

Figure 20-4: Accessing remote interfaces from the client

Each of the variables in the code fragment denotes an interface. However, because the code fragment is on the client, we know that the actual objects held by these variables are from the stub classes. Note that the fragment doesn't show how variable *rdvr* obtains its stub; we will postpone that issue until we cover the RMI registry.

Consider the call to *rdvr.connect*. The stub implements this method by sending a request over the network to its corresponding *RemoteDriver* implementation object on the server. This remote implementation object executes its *connect* method on the server, which will cause a new *RemoteConnection* implementation object to be created on the server. A stub for this new remote object is sent back to the client, which stores it as the value of variable *rconn*.

Now consider the call to *rconn.createStatement*. The stub object sends a request to its corresponding *RemoteConnection* implementation object back on the server. This remote object executes its *createStatement* method. A *RemoteStatement* implementation object gets created on the server, and its stub is returned to the client.

20.2.2 The RMI Registry

Each client-side stub object contains a reference to its corresponding server-side remote implementation object. A client, once it has a stub object, is able to interact with the server through this object, and that interaction may create other stub objects for the client to use. But the question remains – how does a client get its first stub? RMI solves this problem by means of a program called the *rmi registry*.

> *A server publishes stub objects in the RMI registry.*
> *Clients retrieve stub objects from the RMI registry.*

The SimpleDB server publishes just one object, of type *RemoteDriver*. The publishing is performed by the following two lines of code from the *simpledb.server.Startup* program:

```
RemoteDriver d = new RemoteDriverImpl();
Naming.rebind("simpledb", d);
```

The call to *Naming.rebind* creates a stub for the remote implementation object *d*, saves it in the rmi registry, and makes it available to clients under the name “simpledb”.

A client can request a stub from the registry by calling the method *Naming.lookup*. In SimpleDB, this request is made by the following two lines in the *SimpleDriver* class:

```
String newurl = url.replace("jdbc:simpledb", "rmi") + "/simpledb";
RemoteDriver rdvr = (RemoteDriver) Naming.lookup(newurl);
```

For example, if the variable *url* has the value “jdbc:simpledb://localhost”, then the value of *newurl* will be “rmi://localhost/simpledb”. This new string is in the form required by the RMI registry. The call to *Naming.lookup* goes to the RMI registry on the specified host (here, *localhost*), retrieves the stub from it named “simpledb”, and returns it to the caller.

20.2.3 Thread Issues

When building a large Java program, it is always a good idea to be very clear about what threads exist at any point. In a client-server execution of SimpleDB, there will be two sets of threads: the threads on the client machines and the threads on the server machine.

Each client has its own thread on its machine. This thread continues throughout the execution of the client; all of a client’s stub objects are called by this thread. On the other hand, each remote object on the server executes in its own separate thread. A server-side remote object can be thought of as a “mini-server”, which sits waiting for its stub to connect to it. When a connection is made, the remote object performs the requested work, sends the return value back to the client, and waits patiently for another connection. The *RemoteDriver* object created by *simpledb.server.Startup* runs in a thread that can be thought of as the “database server” thread.

Whenever a client makes a remote method call, the client thread waits while the corresponding server thread runs, and resumes when the server thread returns a value. Similarly, the server-side thread will be dormant until one of its methods is called, and will resume its dormancy when the method completes. Thus only one of these client and server threads will be doing anything at any given time. Informally, it seems as if the client’s thread as actually moving back and forth between the client and server as remote calls are made. Although this image can help you visualize the flow of control in a client-server application, it is also important to understand what is really happening.

One way to distinguish between the client-side and server-side threads is to print something. A call to *System.out.println* will show up on the client machine when called from a client thread, and on the server machine when called from a server thread.

20.3 Implementing the Remote Interfaces

The implementation of each remote interface requires two classes: the stub class and the remote implementation class. By convention, the name of the remote implementation class is its interface name appended with the suffix “Impl”, and the name of the stub class is the interface name appended with “Impl_Stub”.

Fortunately, the communication between client-side and server-side objects is the same for all remote interfaces, which means that all of the communication code can be provided by the RMI library classes. The programmer only needs to supply the code that is specific to each particular interface. In other words, the programmer does not need to write the stub classes at all, and writes only the portions of the remote implementation classes that specify what the server does for each method call. The following subsections describe SimpleDB's implementation of each of its remote interfaces.

20.3.1 The class *RemoteDriverImpl*

The class *RemoteDriverImpl* is the entry point into the server; its code appears in Figure 20-5. There will be only one *RemoteDriverImpl* object created, by the *simpledb.server.Startup* bootstrap class, and its stub is the only object published in the RMI registry. Each time its *connect* method is called (via the stub), it creates a new *RemoteConectionImpl* remote object on the server and runs it in a new thread. RMI transparently creates the corresponding *RemoteConnection* stub object and returns it to the client.

```
public class RemoteDriverImpl extends    UnicastRemoteObject
                              implements RemoteDriver {

   public RemoteDriverImpl() throws RemoteException {
   }

   public RemoteConnection connect() throws RemoteException {
      return new RemoteConnectionImpl();
   }
}
```

Figure 20-5: The SimpleDB class *RemoteDriverImpl*

Note how this code is concerned only with server-side objects. In particular, it contains no network code or references to its associated stub object, and when it needs to create a new remote object, it only creates the remote implementation object (and not the stub object). The RMI class *UnicastRemoteObject* contains all of the code needed to perform these other tasks.

20.3.2 The class *RemoteConnectionImpl*

The class *RemoteConnectionImpl* manages client connections on the server; its code appears in Figure 20-6. Each client connection has an associated *RemoteConnectionImpl* object running on the server, in its own thread. This object manages transactions and creates statements for the client. Most of the work of the object is performed by the *Transaction* object *tx*.

```
class RemoteConnectionImpl extends UnicastRemoteObject
                           implements RemoteConnection {
  private Transaction tx;

  RemoteConnectionImpl() throws RemoteException {
    tx = new Transaction();
  }

  public RemoteStatement createStatement() throws RemoteException {
    return new RemoteStatementImpl(this);
  }

  public void close() throws RemoteException {
    tx.commit();
  }

// The following methods are used by the server-side classes.

  Transaction getTransaction() {
    return tx;
  }

  void commit() {
    tx.commit();
    tx = new Transaction();
  }

  void rollback() {
    tx.rollback();
    tx = new Transaction();
  }
}
```

Figure 20-6: The SimpleDB class *RemoteConnectionImpl*

The package-private methods *getTransaction*, *commit*, and *rollback* are not part of the remote interface, and cannot be called by a client. They are used locally by the server-side classes *RemoteStatementImpl* and *RemoteResultSetImpl*.

20.3.3 The class *RemoteStatementImpl*

The class *RemoteStatementImpl* is responsible for executing SQL statements; its code appears in Figure 20-7. The method *executeQuery* obtains a plan from the planner, and passes the plan to a new *RemoteResultSet* object for execution. The method *executeUpdate* simply calls the planner's corresponding method.

```
class RemoteStatementImpl extends UnicastRemoteObject
                          implements RemoteStatement {
  private RemoteConnectionImpl rconn;

  public RemoteStatementImpl(RemoteConnectionImpl rconn)
                                          throws RemoteException {
    this.rconn = rconn;
  }

  public RemoteResultSet executeQuery(String qry)
                                          throws RemoteException {
    try {
      Transaction tx = rconn.getTransaction();
      Plan pln = SimpleDB.planner().createQueryPlan(qry, tx);
      return new RemoteResultSetImpl(pln, rconn);
    }
    catch(RuntimeException e) {
      rconn.rollback();
      throw e;
    }
  }

  public int executeUpdate(String cmd) throws RemoteException {
    try {
      Transaction tx = rconn.getTransaction();
      int result = SimpleDB.planner().executeUpdate(cmd, tx);
      rconn.commit();
      return result;
    }
    catch(RuntimeException e) {
      rconn.rollback();
      throw e;
    }
  }
}
```

Figure 20-7: The SimpleDB code for the class *RemoteStatementImpl*

These two methods are also responsible for implementing the JDBC autocommit semantics. If the SQL statement executes correctly, then it must get committed. The method *executeUpdate* tells the connection to commit the current transaction as soon as the update statement has completed. On the other hand, the method *executeQuery* cannot immediately commit, because its result set is still in use. Instead, the result set's *close* method will commit the transaction.

If something goes wrong during the execution of an SQL statement, then the planner code will throw a runtime exception. These two methods will catch this exception and roll back the transaction.

20.3.4 The class *RemoteResultSetImpl*

The class *RemoteResultSetImpl* contains methods for executing a query plan; its code appears in Figure 20-8. Its constructor opens the plan object given to it, and saves the resulting scan. The methods *next*, *getInt*, *getString*, and *close* simply call their corresponding scan methods. The method *close* also commits the current transaction, as required by the JDBC autocommit semantics.

```
class RemoteResultSetImpl extends UnicastRemoteObject
                          implements RemoteResultSet {
  private RemoteConnectionImpl rconn;

  public RemoteResultSetImpl(Plan plan, RemoteConnectionImpl rconn)
                                                 throws RemoteException {
    s = plan.open();
    sch = plan.schema();
    this.rconn = rconn;
  }

  public boolean next() throws RemoteException {
    return s.next();
  }

  public int getInt(String fldname) throws RemoteException {
    fldname = fldname.toLowerCase(); // to ensure case-insensitivity
    return s.getInt(fldname);
  }

  public String getString(String fldname) throws RemoteException {
    fldname = fldname.toLowerCase(); // to ensure case-insensitivity
    return s.getString(fldname);
  }

  public RemoteMetaData getMetaData() throws RemoteException {
    return new RemoteMetaDataImpl(sch);
  }

  public void close() throws RemoteException {
    s.close();
    rconn.commit();
  }
```

Figure 20-8: The code for the SimpleDB class *RemoteResultSet*

The *RemoteResultSetImpl* class obtains a *Schema* object from its plan. This *Schema* object contains the metadata needed by the *RemoteMetaDataImpl* object, as we shall see next.

20.3.5 The class RemoteMetaDataImpl

A *RemoteMetaDataImpl* object contains the *Schema* object that was passed into its constructor; its code appears in Figure 20-9. The class *Schema* contains analogous methods to those in *RemoteMetaDataImpl*; the difference is that the *RemoteMetaDataImpl* methods refer to fields by column number whereas the *Schema* methods refer to fields by name. The code for *RemoteMetaDataImpl* therefore involves translating the method calls from one way to the other.

```
public class RemoteMetaDataImpl extends UnicastRemoteObject
                                implements RemoteMetaData {
  private Schema sch;
  private Object[] fieldnames;

  public RemoteMetaDataImpl(Schema sch) throws RemoteException {
    this.sch = sch;
    fieldnames = sch.fields().toArray();
  }

  public int getColumnCount() throws RemoteException {
    return fieldnames.length;
  }

  public String getColumnName(int column) throws RemoteException {
    return (String) fieldnames[column-1];
  }

  public int getColumnType(int column) throws RemoteException {
    String fldname = getColumnName(column);
    return sch.type(fldname);
  }

  public int getColumnDisplaySize(int column) throws RemoteException {
    String fldname = getColumnName(column);
    int    fldtype = sch.type(fldname);
    int  fldlength = sch.length(fldname);
    if (fldtype == INTEGER)
      return 6;  // accommodate 6-digit integers
    else
      return fldlength;
  }
}
```

Figure 20-9: The code for the SimpleDB class *RemoteMetaDataImpl*

20.4 Implementing the JDBC Interfaces

SimpleDB's implementation of the RMI remote classes provide all of the features required by the JDBC interfaces in *java.sql*, except two: The RMI methods do not throw SQL exceptions, and they do not implement all of the methods in the interface. That is, we have viable classes that implement interfaces *RemoteDriver*, *RemoteConnection*, etc., but what we really need are classes that implement *Driver*, *Connection*, etc. This is a common problem in object-oriented programming; the solution is to implement the required classes as client-side *wrappers* of their corresponding stub objects.

To see how the wrapping works, let's start with the class *SimpleDriver*, whose code appears in Figure 20-10. The method *connect* must return an object of type *Connection*, which in this case will be a *SimpleConnection* object. To do so, it first obtains a *RemoteDriver* stub from the RMI registry. It then calls the stub's *connect* method to obtain a *RemoteConnection* stub. The desired *SimpleConnection* object is created by passing the *RemoteConnection* stub into its constructor.

```
public class SimpleDriver extends DriverAdapter implements Driver {
  public Connection connect(String host, Properties prop)
                                                    throws SQLException {
     try {
       String newurl = url.replace("jdbc:simpledb","rmi") + "/simpledb";
       RemoteDriver rdvr = (RemoteDriver) Naming.lookup(newurl);
       RemoteConnection rconn = rdvr.connect();
       return new SimpleConnection(rconn);
     }
     catch (Exception e) {
       throw new SQLException(e);
     }
  }
}
```

Figure 20-10: The code for the SimpleDB class *SimpleDriver*

Now take a look at the class *SimpleConnection* in Figure 20-11. The *RemoteConnection* stub object that is passed into its constructor is saved. Calls to methods *createStatement* and *close* are all simply passed onward to the saved object (and from there to the server). This is what is meant by "wrapping" – the saved *RemoteConnection* stub object is doing all of the work, and the *SimpleConnection* object is just acting as a wrapper surrounding it. The primary job of this wrapper is to intercept exceptions thrown by the *RemoteConnection* stub. Whenever the *RemoteConnection* object throws an exception, the *SimpleConnection* wrapper catches the exception and throws an *SQLException* instead.

```
public class SimpleConnection extends ConnectionAdapter
                              implements Connection {
  private RemoteConnection rconn;

  public SimpleConnection(RemoteConnection c) {
    rconn = c;
  }

  public Statement createStatement() throws SQLException {
    try {
      RemoteStatement rstmt = rconn.createStatement();
      return new SimpleStatement(rstmt);
    }
    catch(Exception e) {
      throw new SQLException(e);
    }
  }

  public void close() throws SQLException {
    try {
      rconn.close();
    }
    catch(Exception e) {
      throw new SQLException(e);
    }
  }
}
```

Figure 20-11: The code for the SimpleDB class *SimpleConnection*

The code for the method *createStatement* illustrates how new wrapper objects get created. The *SimpleConnection* object passes the method call onward to the wrapped *RemoteConnection* stub, which returns a *RemoteStatement* stub object. This object is then passed into the constructor of *SimpleStatement*, and that new wrapper object is returned. The code for the classes *SimpleStatement*, *SimpleResultSet*, and *SimpleMetaData* are all similar, and are omitted.

The *SimpleDriver* code in Figure 20-10 also illustrates the transformation of the protocol from the connection URL. JDBC requires that the protocol be of the form “jdbc:xxx:”, where *xxx* identifies the driver. In SimpleDB, the string is always “jdbc:simpledb:”. The driver code transforms this protocol to “rmi:”, which is used by RMI.

We need to discuss one more issue: How can *SimpleDriver* implement the JDBC *Driver* interface, if it doesn’t implement all of the methods in *Driver*? The answer lies with the class *DriverAdapter*, whose code appears in Figure 20-12. This class implements all of the *Driver* methods, but none of the methods do anything – they either throw an *SQLException* saying that the method is not supported, or they return a default value.

```
public abstract class DriverAdapter implements Driver {

   public boolean acceptsURL(String url) throws SQLException {
      throw new SQLException("operation not implemented");
   }

   public Connection connect(String url, Properties info)
                                         throws SQLException {
      throw new SQLException("operation not implemented");
   }

   public int getMajorVersion() {
      return 0;
   }

   public int getMinorVersion() {
      return 0;
   }

   public DriverPropertyInfo[] getPropertyInfo(String url,
                                                Properties info) {
      return null;
   }

   public boolean jdbcCompliant() {
      return false;
   }
}
```

Figure 20-12: The code for the SimpleDB class *DriverAdapter*

Note that when *SimpleDriver* extends the *DriverAdapter* class, any methods defined in *SimpleDriver* will override the corresponding *DriverAdapter* methods. The result is that *SimpleDriver* actually implements the entire *Driver* interface, with the property that the methods that are not specifically in the *SimpleDriver* class will throw exceptions.

20.5 Chapter Summary

- There are two ways that an application program can access a database:
    - It can access the database indirectly, via a server.
    - It can access the database directly, by *embedding* the database system.

- The "embedded database system" strategy is appropriate for applications that use their database privately, such as a GPS navigation system or sensor monitor. Often, an embedded database system can be customized for its application by removing unnecessary features.

- SimpleDB implements client-server communication via the Java *Remote Method Invocation* (*RMI*) mechanism. Each JDBC interface has a corresponding RMI remote interface. Their primary difference is that they throw RemoteExceptions (as required by RMI) instead of SQLExceptions (as required by JDBC).

- Each server-side remote implementation object executes in its own thread, waiting for a stub to contact it. The SimpleDB startup code creates a remote implementation object of type *RemoteDriver*, and stores a stub to it in the *RMI registry*. When a JDBC client wants a connection to the database system, it obtains the stub from the registry and calls its *connect* method.

- The *connect* method is typical of RMI remote methods. It creates a new *RemoteConnectionImpl* object on the server machine, which runs in its own thread. The method then returns a stub to this object back to the JDBC client. The client can call *Connection* methods on the stub, which cause the corresponding methods to be executed by the server-side implementation object.

- JDBC clients do not use remote stubs directly, because they implement the remote interfaces instead of the JDBC interfaces. Instead, the client-side objects *wrap* their corresponding stub objects.

20.6 Suggested Reading

There are numerous books dedicated to explaining RMI, such as [Grosso 2001]. In addition, SUN’s website at *http://java.sun.com/products/jdk/rmi/reference/docs/index.html* provides tutorials and other “official” documentation devoted to RMI.

The driver implementation used by SimpleDB is technically known as a “Type 4” driver. The online article [Nanda 2002] describes and compares the four different driver types. The companion online article [Nanda et al. 2002] leads you through the construction of an analogous Type 3 driver.

20.7 Exercises

CONCEPTUAL EXERCISES

20.1 Trace the code of the demo client *StudentMajor.java*, using the code from the classes in *simpledb.remote*. What server-side objects get created? What client-side objects get created? What threads get created?

20.2 The *RemoteStatementImpl* methods *executeQuery* and *executeUpdate* require a transaction. A *RemoteStatementImpl* object gets its transaction by calling *rconn.getTransaction()* each time *executeQuery* or *executeUpdate* is called. A simpler strategy would be to just pass the transaction to each *RemoteStatementImpl* object when it was created, via its constructor. However, this would be a very bad idea. Give a scenario in which something incorrect could happen.

20.3 We know that remote implementation objects live on the server. But are the remote implementation classes needed by the client? Are the remote interfaces needed by the client? Create a client configuration that contains the SimpleDB folders *sql* and *remote*. What class files can you remove from these folders without causing the client to break? Explain your results.

PROGRAMMING EXERCISES

20.4 Revise the SimpleDB code so that it implements the following methods of *ResultSet*.
a) The method *beforeFirst()*, which repositions the result set to before the first record (i.e., to its original state). Use the fact that scans have a *beforeFirst* method which does the same thing.
b) The method *absolute(int n)*, which positions the result set to the nth record. (Scans do not have a corresponding *absolute* method.)

20.5 Exercise 17.19 asked you to implement the scan methods *afterLast* and *previous*.
a) Modify the *ResultSet* interface and implementation to contain these methods.
b) Test your code by modifying the demo JDBC client class *SQLInterpreter* to print its output tables in reverse order.

20.6 The JDBC *Statement* interface contains a method *close()*, which closes any result set for that statement that may still be open. Implement this method.

20.7 Standard JDBC specifies that the method *Connection.close* should close all of its statements (as in Exercise 20.6). Implement this feature.

20.8 Standard JDBC specifies that a connection is automatically closed when a *Connection* object is garbage collected (e.g., when a client program completes). This ability is important, as it allows the database system to release resources that were abandoned by forgetful clients. Use the *finalizer* construct in Java to implement this feature.

20.9 SimpleDB implements autocommit mode, in which the system automatically decides when to commit a transaction. Standard JDBC allows the client to turn off autocommit mode, and to commit and rollback its transactions explicitly. The JDBC *Connection* interface has a method *setAutoCommit(boolean ac),* which allows a client to turn auto-commit mode on or off, a method *getAutoCommit*, which returns the current auto-commit status, and the methods *commit* and *rollback*. Implement these methods.

20.10 The SimpleDB server allows anyone to connect to it. Modify class *SimpleDriver* so that its *connect* method authenticates users. The method should extract a username and password from the *Properties* object passed into it. The method should then compare them against the contents of a server-side text file, and throw an exception if there is no match. Assume that new usernames and passwords are added (or dropped) by simply editing the file on the server.

20.11 Modify *RemoteConnectionImpl* so that it only allows a limited number of connections at a time. What should the system do if there are no available connections left when a client tries to connect?

20.12 In JDBC, the package *java.sql* contains an interface *PreparedStatement*, which allows JDBC clients to separate the planning stage of a query from the execution of its scan. A query can be planned once and executed multiple times, perhaps with different values for some of its constants. Consider the following code fragment:

```
String qry = "select SName from STUDENT where MajorId = ?";
PreparedStatement ps = conn.prepareStatement(qry);
ps.setInt(1, 20);
ResultSet rs = ps.executeQuery();
```

The "?" character in the query denotes an unknown constant, whose value will be assigned prior to execution. A query can have multiple unknown constants. The method *setInt* (or *setString*) assigns a value to the ith unknown constant.
a) Suppose that the prepared query contains no unknown constants. Then the *PreparedStatement* constructor obtains the plan from the planner, and the *executeQuery* method passes the plan to the *ResultSet* constructor. Implement this special case, which involves changes to the *sql* and *remote* packages, but no changes to the parser or planner.
b) Now revise your implementation so that it handles unknown constants. The parser must be changed to recognize the "?" characters. The planner must be able to obtain a list of the unknown constants from the parser; those constants can then be assigned values via the *setInt* and *setString* methods.

20.13 Suppose you start up a JDBC client program; however, it takes too long to finish, so you cancel it using <CTRL-C>.
a) What impact does this have on the other JDBC clients running on the server?
b) When and how will the server notice that your JDBC client program is no longer running? What will it do when it finds out?
c) What is the best way for the server to handle this kind of situation?
d) Design and implement your answer to (c).

20.14 Write a Java class *Shutdown* whose main method gracefully shuts down the server. That is, existing connections are allowed to complete, but no new connections should be made. When there are no more transactions running, the code should write a quiescent checkpoint record to the log and write an "ok to shut down" message on the console. (Hint: The easiest way to do this is to remove the *DbServer* object from the RMI registry. Also, remember that this method will execute in a different JVM from the server. You therefore will need to modify the server somehow so that it recognizes that *Shutdown* has been called.)

PART 4
Efficient Query Processing

The focus of Part 3 was to create a simple yet functional database server. Unfortunately, the simplicity of SimpleDB comes with a cost – the plans that it generates are remarkably inefficient. SimpleDB may work tolerably for small databases, but a multi-table query over a large database could easily take years to execute. This situation is clearly unacceptable for any database system that hopes to be practical. In Part 4 we tackle this efficiency issue.

There are two fundamental reasons why SimpleDB query processing is ineffective:

- Its operator implementations are too inefficient.
- Its planner is too naïve.

The next four chapters consider these liabilities. Chapters 21-23 consider three techniques for implementing relational operators efficiently: indexing, the materialization and sorting of query output, and making better use of buffers. Chapter 24 then reconsiders the query planning process. We show how the planner can determine an efficient plan for an SQL query, choosing from the relational algebra query trees equivalent to it, and the various implementations for each relational operator in the tree.

21
INDEXING
and the packages in simpledb.index

When we write a query that mentions a table, we may be interested in only a few of its records, namely the records satisfying the selection predicate or the records joining with a particular record. An *index* is a file that allows the database system to locate these records without having to search through the entire table. In this chapter we consider three common ways to implement an index: static hashing, extendable hashing, and B-trees. We then develop implementations of the relational algebra operations *select* and *join* that can take advantage of indexes.

21.1 The Index *interface*

Chapter 6 introduced the basic principles of indexing, which can be summarized as follows: An *index* is a file that is organized on some fields of a table. Each index record corresponds to a data record. Each index record contains a value for the indexed fields, plus the record identifier (or *rid*) of its corresponding data record. When the database system wants to find the data records that have a particular value for the indexed fields, it can go to the index, find the specified index records, and use the rids stored in those index records to move to the desired data records. This strategy is fast, and avoids having to search through the entire table.

In this chapter we examine how a database system such as SimpleDB implements and uses indexes. In SimpleDB, an index can have only one indexed field, which means that an index record consists of two values. We call these values the *dataval* and the *datarid*.

> *Each SimpleDB index record contains a dataval and a datarid.*
> - *The dataval is the value of the indexed field.*
> - *The datarid is the rid of the corresponding data record.*

An index in SimpleDB is an object in a class that implements the interface *Index*; see Figure 21-1.

```
public interface Index {
      public void    beforeFirst(Constant searchkey);
      public boolean next();
      public RID     getDataRid();
      public void    insert(Constant dataval, RID datarid);
      public void    delete(Constant dataval, RID datarid);
      public void    close();
}
```

Figure 21-1: The code for the SimpleDB *Index* interface

Many of the methods in *Index* are similar to methods in *RecordFile*: A client can position the index at the beginning and move through its records, can retrieve the contents of the current index record, and can insert and delete index records. However, because indexes are used in well-known, specific ways, some of the methods in *Index* are more specific than those in *RecordFile*.

A SimpleDB client always searches an index by providing a value (called the *search key*) and retrieving the index records having a matching dataval. The method *beforeFirst* takes this search key as its argument. Subsequent calls to *next* move the index to the next record whose dataval equals the search key, and return *false* if no more such records exist.

An index does not need general-purpose *getInt* and *getString* methods, because all index records have the same two fields. Moreover, a client never needs to retrieve a record's dataval, because it will always be the search key. Thus the only retrieval method it needs is *getDataRid*, which returns the datarid of the current index record.

The SimpleDB code fragment of Figure 21-2 provides an example of index use. The code implements the algorithm of Figure 6-11 to search for all students having major #10, and it prints the names of these students. Note that the code uses a table scan to retrieve the STUDENT records, even though the table is not really "scanned". Instead, the code calls the table scan's *moveToRid* method to position the scan at the desired record.

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();

// Open a scan on the data table.
Plan studentplan = new TablePlan("student", tx);
TableScan studentscan = (TableScan) studentplan.open();

// Open the index on MajorId.
MetadataMgr mdmgr = SimpleDB.mdMgr();
Map<String,IndexInfo> indexes = mdmgr.getIndexInfo("student", tx);
IndexInfo ii = indexes.get("majorid");
Index idx = ii.open();

// Retrieve all index records having a dataval of 10.
idx.beforeFirst(new IntConstant(10));
while (idx.next()) {
    // Use the datarid to go to the corresponding STUDENT record.
    RID datarid = idx.getDataRid();
    studentscan.moveToRid(datarid);
    System.out.println(studentscan.getString("sname"));
}

// Close the index and the data table.
idx.close();
studentscan.close();
tx.commit();
```

Figure 21-2: Using an index in SimpleDB

The code fragment of Figure 21-3 illustrates how the database system deals with updates to a data table. The code performs two tasks. The first task inserts a new record into STUDENT; the second task deletes a record from STUDENT. The code must deal with the insertion by inserting a corresponding record in each index for STUDENT, and similarly for the deletion. Note how the code begins by opening all of the indexes for STUDENT, and saving them in a map. The code can then loop through this map each time it needs to do something to each index.

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();
Plan studentplan = new TablePlan("student", tx);
TableScan studentscan = (TableScan) studentplan.open();

// Create a map containing all indexes for STUDENT.
MetadataMgr mdmgr = SimpleDB.mdMgr();
Map<String,Index> indexes = new HashMap<String,Index>();
Map<String,IndexInfo> idxinfo = mdmgr.getIndexInfo("student", tx);
for (String fldname : idxinfo.keySet()) {
   Index idx = idxinfo.get(fldname).open();
   indexes.put(fldname, idx);
}

// Task 1: insert a new STUDENT record for Sam
//    First, insert the record into STUDENT.
studentscan.insert();
studentscan.setInt("sid", 11);
studentscan.setString("sname", "sam");
studentscan.setInt("gradyear", 2010);
studentscan.setInt("majorid",  30);

//    Then insert a record into each of the indexes.
RID datarid = studentscan.getRid();
for (String fldname : indexes.keySet()) {
   Constant dataval = studentscan.getVal(fldname);
   Index idx = indexes.get(fldname);
   idx.insert(dataval, datarid);
}

// Task 2: find and delete Joe's record
studentscan.beforeFirst();
while (studentscan.next()) {
   if (studentscan.getString("sname").equals("joe")) {

      // First, delete the index records for Joe.
      RID joeRid = studentscan.getRid();
      for (String fldname : indexes.keySet()) {
         Constant dataval = studentscan.getVal(fldname);
         Index idx = indexes.get(fldname);
         idx.delete(dataval, datarid);
      }

      // Then delete Joe's record in STUDENT.
      studentscan.delete();
      break;
   }
}
studentscan.close();
for (Index idx : indexes.values())
   idx.close();
tx.commit();
```

Figure 21-3: Updating indexes to reflect changes to data records

The code of Figures 21-2 and 21-3 manipulate indexes without knowing (or caring) how they are actually implemented. The only requirement is that the indexes implement the *Index* interface. The remainder of the chapter is organized as follows. Sections 21.2-21.4 examine three rather different kinds of index implementation – two strategies based on hashing, and a strategy based on sorted trees. Sections 21.5 and 21.6 then consider how the query processor and planner can take advantage of indexes.

21.2 Static Hash Indexes

We begin with static hashing, which is probably the simplest way to implement an index. Although it is not the most efficient strategy, it is easy to understand and illustrates the principles most clearly. Thus it is a good place to begin.

21.2.1 Static hashing

A static hashed index uses a fixed number N of *buckets*, numbered from 0 to N-1. The index also uses a *hash function*, which maps values to buckets. Each index record is assigned to the bucket obtained from hashing its dataval. Although records with different datavals can be assigned to the same bucket, the hash function attempts to distribute the records equally among the buckets.

> *A static hash index stores index records in buckets.*
> *Records having the same hash value are stored in the same bucket.*

A static hashed index works as follows:

- To store an index record, put it into the bucket assigned by the hash function.
- To find an index record, hash the search key and examine that bucket.
- To delete an index record, first find it (as above), and then delete it from the bucket.

The search cost of a hashed index is related to how many buckets it has. If an index contains B blocks and has N buckets, then each bucket is about B/N blocks long, and so searching a bucket requires about B/N block accesses.

For example, consider an index on *SName.* Suppose for simplicity that N=3 and that the hash function maps a string s to the number of letters in s (mod N) that come earlier in the alphabet than 'm'.† Assume also that 3 index records fit into a block. Figure 21-4 depicts the contents of the three index buckets. In the figure, we use r_i to denote the rid of the i^{th} STUDENT record.

† This is a remarkably bad hash function, but it helps make the example interesting.

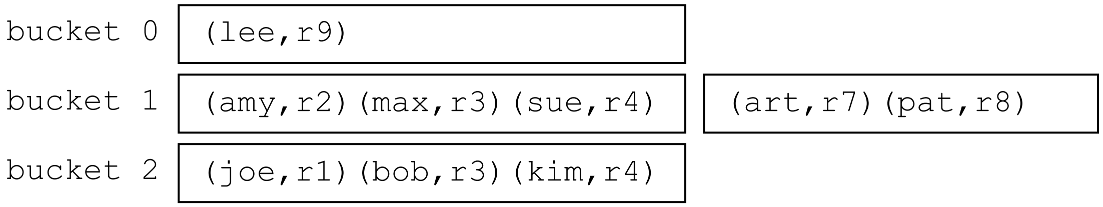


Figure 21-4: A static hash index with three buckets

Suppose now that we want to find the datarid of all students named "sue". We hash the string "sue" to get bucket 1, and search that bucket. The search requires 2 block accesses. Similarly, since "ron" hashes to bucket 0, it takes just 1 block access to determine that there are no students named "ron".

This example uses ridiculously small values for the block size and number of buckets. For a more realistic sample calculation, assume that the index uses 1024 buckets, which means (assuming that the records hash evenly among the buckets) that:

- an index of up to 1024 blocks can be searched in only 1 disk access;
- an index of up to 2048 blocks can be searched in only 2 disk accesses;

and so on. To give some perspective to these numbers, note that an index record on *SName* requires 22 bytes (14 bytes for the *varchar(10)* dataval, and 8 bytes for the datarid); so if we add 1 byte per record to hold the EMPTY/INUSE flag, then 178 index records will fit into a 4K block. An index size of 2048 blocks therefore corresponds to a data file of about 364,544 records. That is a lot of records to be able to search in only 2 disk accesses!

21.2.2 Implementing static hashing

Static hashing in SimpleDB is implemented in the class *HashIndex*, whose code appears in Figure 21-5.

```
public class HashIndex implements Index {
   public static int NUM_BUCKETS = 100;
   private String idxname;
   private Schema sch;
   private Transaction tx;
   private Constant searchkey = null;
   private TableScan ts = null;

   public HashIndex(String idxname, Schema sch, Transaction tx) {
      this.idxname = idxname;
      this.sch = sch;
      this.tx = tx;
   }

    public void beforeFirst(Constant val) {
      close();
      searchkey = val;
      int bucket = val.hashCode() % NUM_BUCKETS;
      String tblname = idxname + bucket;
      TableInfo ti = new TableInfo(tblname, sch);
      ts = new TableScan(ri, tx);
    }

   public boolean next() {
      while (ts.next())
         if (ts.getVal("dataval").equals(searchkey))
            return true;
      return false;
   }

   public RID getDataRid() {
      int blknum = ts.getInt("block");
      int id = ts.getInt("id");
      return new RID(blknum, id);
   }

   public void insert(Constant val, RID rid) {
      beforeFirst(val);
      ts.insert();
      ts.setInt("block", rid.blockNumber());
      ts.setInt("id", rid.id());
      ts.setVal("dataval", val);
   }

   public void delete(Constant val, RID rid) {
      beforeFirst(val);
      while(next())
         if (getDataRid().equals(rid)) {
            ts.delete();
            return;
         }
   }

    public void close() {
        if (ts != null)
         ts.close();
    }

   public static int searchCost(int numblocks, int rpb) {
      return numblocks / HashIndex.NUM_BUCKETS;
   }
}
```

Figure 21-5: The code for the SimpleDB class *HashIndex*

This class stores each bucket in a separate table, whose name is the catenation of the index name and the bucket number. For example, the table for bucket #35 of index SID_INDEX is named "*SID_INDEX35*". The method *beforeFirst* hashes the search key and opens a table scan for the resulting bucket. Method *next* starts from the current position in the scan, and reads records until one is found having the search key; if no such record is found, the method returns *false*. The datarid of an index record is stored as two integers, in fields *Block* and *Id*. Method *getDataRid* reads these two values from the current record and constructs the rid; the method *insert* does the opposite.

In addition to implementing the methods for the *Index* interface, the class *HashIndex* implements a static method *searchCost*. This method is called by *IndexInfo.blocksAccessed*, as we saw in Figure 16-15. The *IndexInfo* object passes in two arguments to the *searchCost* method: the number of blocks in the index, and the number of index records per block. It does so because it does not know how the indexes compute their costs. In the case of static indexing, the search cost depends only on the index size, and thus the RPB value is ignored.

21.3 Extendable Hash Indexes

The search cost of a static hash index is inversely proportional to the number of buckets – the more buckets we use, the fewer blocks in each bucket. The best possible situation would be to have enough buckets so that each bucket would be exactly one block long.

If an index always stayed the same size, then it would be easy to calculate this ideal number of buckets. But in practice, indexes grow as new records are inserted into the database. So how do we decide how many buckets to use? Suppose that we choose the buckets based on the current index size; then as the index grows, each bucket will eventually wind up containing many blocks. But if we choose a larger number of buckets based on future needs, then the current empty and nearly-empty buckets will create a lot of wasted space until the index grows into it.

A strategy known as *extendable hashing* solves this problem by using a very large number of buckets, guaranteeing that each bucket will never be more than one block long[†]. Extendable hashing deals with the problem of wasted space by allowing multiple buckets to share the same block. The idea is that even though there are a lot of buckets, they all share a small number of blocks, so there is very little wasted space. It's a very clever idea.

The sharing of blocks by buckets is achieved by means of two files: the *bucket file* and the *bucket directory*. The bucket file contains the index records. The bucket directory maps buckets to blocks. The directory can be thought of as an array *Dir* of integers, one integer for each bucket. If an index record hashes to bucket *b*, then that record will be stored in block *Dir[b]* of the bucket file.

For an example, Figure 21-6 depicts the possible contents of an extendable hash index on field *SId* of STUDENT, where we assume (for the sake of readability) that:

- three index records fit into a block,
- only eight buckets are used,
- the hash function h(x) = x mod 8, and
- the STUDENT table contains seven records, having ID 1, 2, 4, 5, 7, 8, and 12.

[†] An exception must be made when too many records have exactly the same dataval. Since those records will always hash to the same block, there will be no way a hashing strategy could spread them across several buckets. In this case, the bucket would have as many blocks as needed to hold those records.

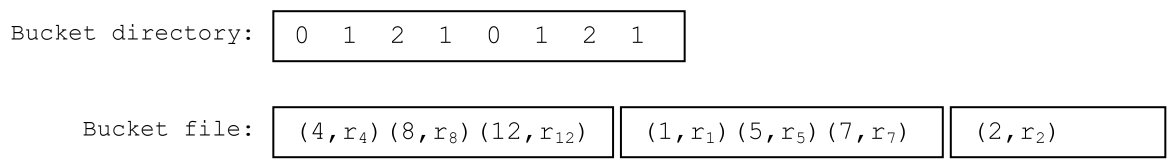


Figure 21-6: An extendable hash index on the field *SId* of STUDENT

As before, we use r_i to denote the rid of the i^{th} STUDENT record.

Note how the bucket directory *Dir* works. In particular, the fact that *Dir[0]*=0 and *Dir[4]*=0 means that records hashing to 0 (such as r_8) or 4 (such as r_4 and r_{12}) will be placed in block 0. Similarly, records hashing to 1, 3, 5, or 7 will be placed in block 1, and records hashing to 2 or 6 will be placed in block 2. This bucket directory thus allows the index records to be stored in three blocks, instead of eight.

Of course, there are many ways to set up the bucket directory to share the buckets among three blocks. The directory shown in Figure 21-6 has a particular logic behind it, which we will discuss next.

21.3.1 Sharing index blocks

Extendable hashed directories always have 2^M buckets; the integer *M* is called the *maximum depth* of the index. A directory of 2^M buckets can support hash values that are *M* bits long. The example of Figure 21-6 used *M*=3. In practice, *M*=32 is a reasonable choice, because integer values have 32 bits.

Initially, an empty bucket file will contain a single block, and all directory entries will point to this block. In other words, this block is shared by all of the buckets. Any new index record will be inserted into this block.

Every block in the bucket file has a *local depth*. A local depth of *L* means that the hash value of every record in the block has the same rightmost *L* bits. The first block of the file initially has a local depth of 0, because its records can have arbitrary hash values.

Suppose that a record is inserted into the index, but does not fit into its assigned block. Then that block *splits* – that is, another block is allocated in the bucket file, and the records in the full block are distributed among itself and the new block. The redistribution algorithm is based on the local depth of the block. Since all records in the block currently have the same rightmost *L* bits of their hash value, we consider the rightmost $(L+1)^{st}$ bit: all records having a 0 are kept in the original block, and all records having a 1 are transferred to the new block. Note that the records in each of these two blocks now have *L*+1 bits in common. That is, the local depth of each block has been increased by 1.

When a block splits, the bucket directory must be adjusted. Let b be the hash value of the newly-inserted index record; that is, b is the number of a bucket. Suppose that the rightmost *L* bits of b are $b_L...b_2b_1$. Then it can be shown (see Exercise 21.6) that the bucket numbers having these rightmost *L* bits (which includes b) all point to the block that just split. Thus we modify the directory so that every slot whose rightmost *L+1* bits are $1b_L...b_2b_1$ points to the new block.

For example, suppose that bucket 17 currently maps to a block *B* having local depth 2. Since 17 in binary is 1001, its rightmost 2 bits are "01". Therefore, we know that the buckets whose rightmost two bits are "01" map to *B*, such as 1, 5, 9, 13, 17, and 21. Now suppose that block *B* is full and needs to split. The system allocates a new block *B′*, and sets the local depth of both *B* and *B′* to 3. It then adjusts the bucket directory. Those buckets whose rightmost 3 bits are "001" continue to map to block *B* (that is, their directory entries stay unchanged). But those buckets whose rightmost 3 bits are "101" are changed to map to *B′*. Thus buckets 1, 9, 17, 25, and so on will continue to map to *B*, whereas buckets 5, 13, 21, 29, and so on now will now map to *B′*.

Figure 21-7 gives the algorithm for inserting a record into an extendable hash index.

```
1.  Hash the record's dataval to get bucket b.
2.  Find $B = Dir[b]$. Let $L$ be the local depth of block $B$.
3a. If the record fits into $B$, insert it and return.
3b. If the record does not fit in $B$:
      • Allocate a new block $B'$ in the bucket file.
      • Set the local depth of both $B$ and $B'$ to be $L+1$.
      • Adjust the bucket directory so that all buckets having the rightmost $L+1$ bits
        $1b_L...b_2b_1$ will point to $B'$.
      • Re-insert each record from $B$ into the index. (These records will hash either
        to $B$ or to $B'$.)
      • Try again to insert the new record into the index.
```

Figure 21-7: The algorithm for inserting a records into an extendable hash index

For an example, consider again an extendable hash index on *SId*. Unlike Figure 21-6, however, we shall use the more realistic assumption that the bucket directory has 2^{10} buckets (that is, the maximum depth is 10), and that the hash function maps each integer n to n%1024. Initially, the bucket file consists of one block, and all directory entries point to that block. The situation is depicted in Figure 21-8(a).

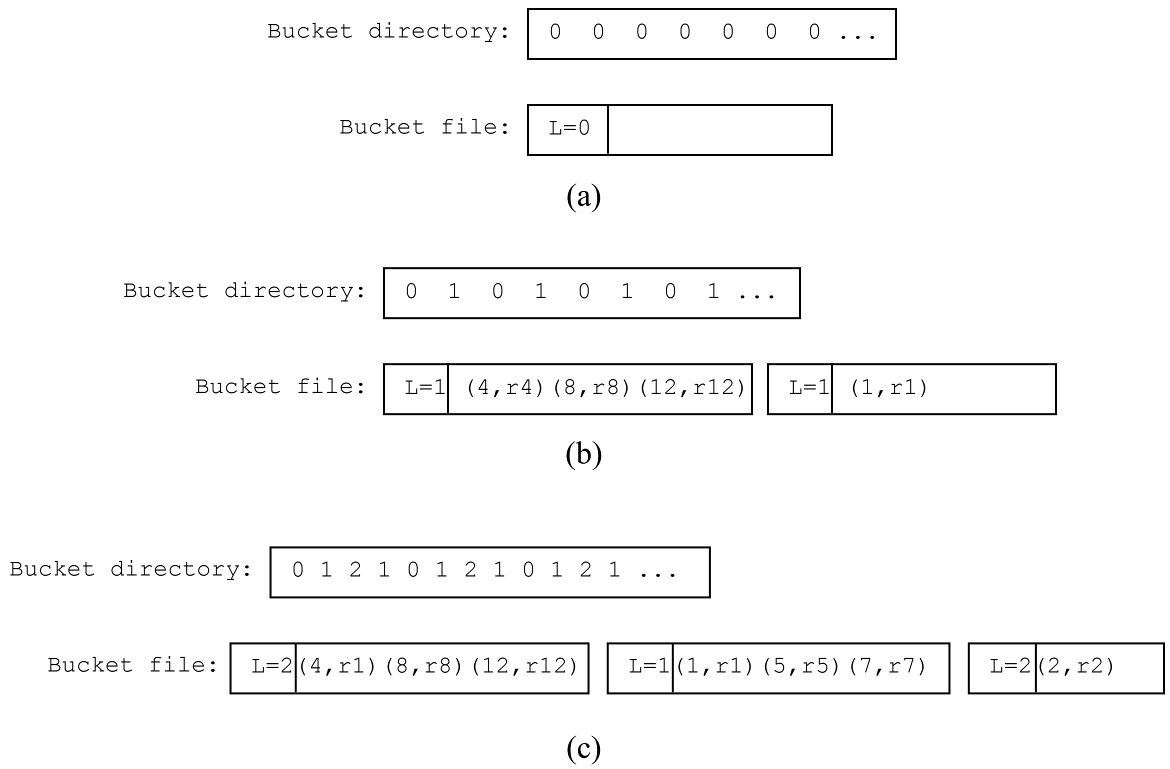


Figure 21-8: Inserting records into an extendable hash index

Suppose now that we insert index records for students 4, 8, 1, and 12. The first three insertions go to block 0, but the fourth one causes a split. This split causes the following events to occur: A new block is allocated, the local depths are increased from 0 to 1, the directory entries are adjusted, the records in block 0 are re-inserted, and the record for employee 12 is inserted. The result is shown in Figure 21-8(b). Note that each odd entry in the bucket directory has been changed to point to the new block. The index is now such that all records having an even hash value (i.e. a rightmost bit of 0) are in block 0 of the bucket file, and all odd-value records (i.e. a rightmost bit of 1) are in block 1.

Next, we insert index records for employees 5, 7 and 2. The first two records fit into block 1, but the third one causes block 0 to split again. The result is shown in Figure 21-8(c). Block 0 of the bucket file now contains all index records whose hash value ends in "00"; and block 2 contains all records whose hash value ends in "10". Block 1 still contains all records whose hash value ends in "1".

One problem with any hashing strategy is that records are not guaranteed to be distributed evenly. When a block splits, all of its records may rehash to the same block; if the new record also hashes to that block, then it still will not fit into the block and the block must be split again. If the local depth ever equals the maximum depth, then no more splitting is possible, and an overflow block must be created to hold the index records.

21.3.2 Compacting the bucket directory

The problem with our description of extendable hashing is the size of the bucket directory. A hash file having a maximum depth of 10 requires a directory of 2^{10} buckets and can be stored in one block, assuming a block size of 4K bytes. However, if we choose a maximum depth of 20, the directory has 2^{20} buckets and requires 1,024 blocks. This directory would be far too large for a small index. The principle of extendable hashing is that the size of the file should expand to fit the size of the index. It turns out that we can also apply this principle to the bucket directory, so that it starts small and grows in conjunction with the index.

The key idea is to note that the bucket directory entries of Figure 21-8 are in a particular pattern. If a block has local depth 1, then every other bucket entry points to that block. If a block has local depth 2, then every fourth bucket entry points to that block. And in general if a block has local depth *L*, then every 2^L bucket entries point to that block. This pattern means that the highest overall local depth determines the "period" of the directory. For example, since the highest local depth in Figure 21-8(c) is 2, the bucket directory contents repeat every 2^2 entries.

The fact that the directory entries repeat means that there is no need to store the entire bucket directory; we only need to store 2^d entries, where *d* is the highest local depth. We call *d* the *global depth* of the index.

The file containing the bucket directory only needs to contain 2^d entries, where d is the highest local depth of any index block.

Our algorithm for searching an index needs to be slightly modified to accommodate this change to the size of the bucket directory. In particular, after the search key is hashed, we only use the rightmost *d* bits of the hash value to determine the bucket directory entry.

Our algorithm for inserting a new index record also needs modification. As with search, we hash the record's dataval and get the bucket directory entry for the rightmost *d* bits of the hash value. We then try to insert the index record into that block. If the block splits, we proceed as usual. The only exception is when the split causes the local depth of the block to become larger than the current global depth of the index. In this case, the global depth must be incremented before the records can be rehashed.

Incrementing the global depth means doubling the size of the bucket directory. Note that doubling the directory is remarkably easy – since the directory entries repeat, the second half of the doubled directory is identical to the first half. Once this doubling has occurred, the splitting process can continue.

To illustrate the algorithm, reconsider the example of Figure 21-8. The initial index will have global depth 0, which means that the bucket directory will have a single entry, pointing to block 0. The insertion of records for 4, 8, and 1 keep the global depth at 0.

Because the global depth is 0, only the rightmost 0 bits of the hash value are used to determine the directory entry; in other words, entry 0 is always used regardless of the hash value. When the record for 12 is inserted, however, the split causes the local depth of block 0 to increase, which means that the global depth of the index must also increase, and the bucket directory doubles from 1 to 2 entries. Initially both entries point to block 0; then all entries whose rightmost 1 bit is "1" are adjusted to point to the new block. The resulting directory has a global depth of 1 and entries *Dir[0]*=0 and *Dir[1]*=1.

Now that the global depth is 1, the insertion of records 5 and 7 use the rightmost 1 bits of the hash value, which is 1 in both cases. Thus bucket *Dir[1]* is used, and both records are inserted into block 1. The split that occurs after record 2 is inserted causes the local depth of block 0 to increase to 2, which means the global depth must also increase. Doubling the directory increases it to four entries, which initially are: 0 1 0 1. The entries having rightmost bits "10" are then adjusted to point to the new block, leading to a directory whose entries are: 0 1 2 1.

Extendable hashing does not work well when the index contains more records with the same dataval than can fit in a block. In this case, no amount of splitting can help, and the bucket directory will expand fully to its maximum size even though there are relatively few records in the index. To avoid this problem, the insertion algorithm must be modified to check for this situation, and create a chain of overflow blocks for that bucket without splitting the block.

21.4 B-tree Indexes

The previous two indexing strategies were based on hashing. We now consider a way to use sorting. The basic idea is this:

> *A sorted index sorts the records in the index file by their datavals.*

21.4.1 How to improve a dictionary

If you think about it, a sorted index file is a lot like a dictionary. An index file is a sequence of index records, each of which contains a dataval and a datarid. A dictionary is a sequence of entries, each of which contains a word and a definition. When we use a dictionary, we want to find the definitions of our word as quickly as possible. When we use an index file, we want to find the datarids of our dataval as quickly as possible. Figure 21-9 summarizes this correspondence.

	Dictionary	**Sorted Index File**
ENTRY	[word, definition] A word can have more than one definition.	[dataval, datarid] A dataval can have more than one datarid.
USAGE	Find the definitions of a given word.	Find the datarids for a given dataval.

Figure 21-9: The correspondence between a dictionary and a sorted index file

The close correspondence between dictionaries and sorted indexes means that we should be able to apply our understanding of dictionaries to the problem of implementing a sorted index. Let's see.

The dictionary on my desk has about 1,000 pages. Each page has a *heading*, which lists the first and last word on that page. When I am looking for a word, the heading helps me find the correct page – I only need to look at the headings, not the contents of the pages. Once I locate the correct page, I then search it to find my word.

The dictionary also has a table of contents, listing the page where the words beginning with each letter begin. However, I never use the table of contents because its information isn't especially useful. What I would really like is for the table of contents to contain one row for each page header, as in Figure 21-10(a). This table of contents is a real improvement, because I no longer have to flip through the pages; all of the header information is in one place.

TABLE OF CONTENTS
page i

word range	page
a – ability	1
abject – abscissa	2
abscond – academic	3
...	

(a)

GUIDE TO THE
TABLE OF CONTENTS

word range	page
a – bouquet	i
bourbon – couple	ii
couplet – exclude	iii
...	

(b)

Figure 21-10: An improved table of contents for a dictionary

A 1,000-page dictionary will have 1,000 headers. If we assume that about 100 headers will fit on a page, then the table of contents will be 10 pages long. Searching through 10 pages is a lot better than searching through 1,000, but it is still too much work. What I need is something that will help me search the table of contents, as in Figure 21-10(b). The "Guide to the Table of Contents" lists the header information for each page in the table of contents. The guide will thus contain 10 headers, and will easily fit on a single page.

With this setup, I could find any word in my dictionary by looking at exactly three pages:

- The guide page tells me which page in the table of contents to use.
- That table of contents page tells me which word-content page to use.
- I then search that word-content page to find my word.

If we try this strategy with a very large dictionary (say, over 10,000 pages), then its table of contents will be over 100 pages long, and the guide will be over one page long. In this case, we could construct a "guide to the guide" page, which would keep us from having to search the guide. In this case, finding a word requires looking at four pages.

If we compare the two parts of Figure 21-10, we see that the table of contents and its guide have exactly the same structure. Let's call these pages the *directory* of the dictionary. The table of contents is the level-0 directory, the guide is the level-1 directory, the guide to the guide is the level-2 directory, and so on.

Thus our improved dictionary has the following structure:

- There are numerous word-content pages, in sorted order.
- Each level-0 directory page contains the header for several word-content pages.
- Each level-(N+1) directory page contains the header for to several level-N directory pages.
- There is a single directory page at the highest level.

This structure can be depicted as a tree of pages, with the highest-level directory page as its root and the word-content pages as its leaves. Figure 21-11 depicts this tree.

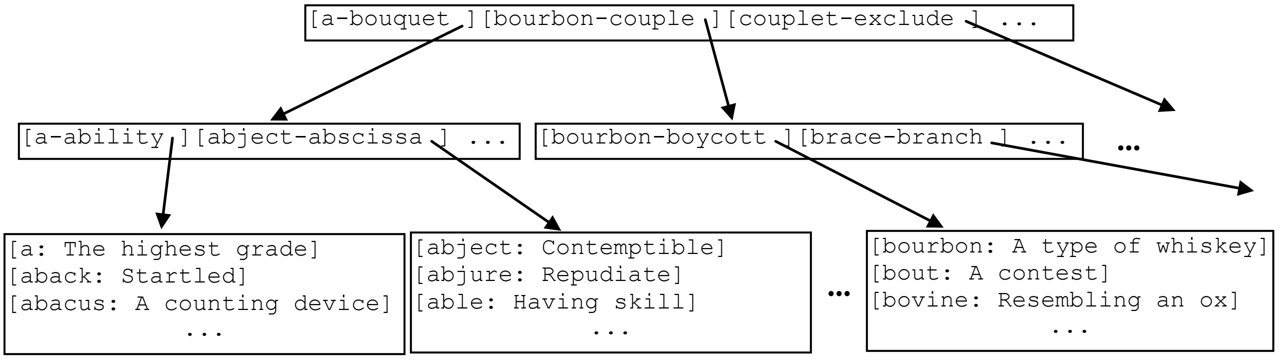
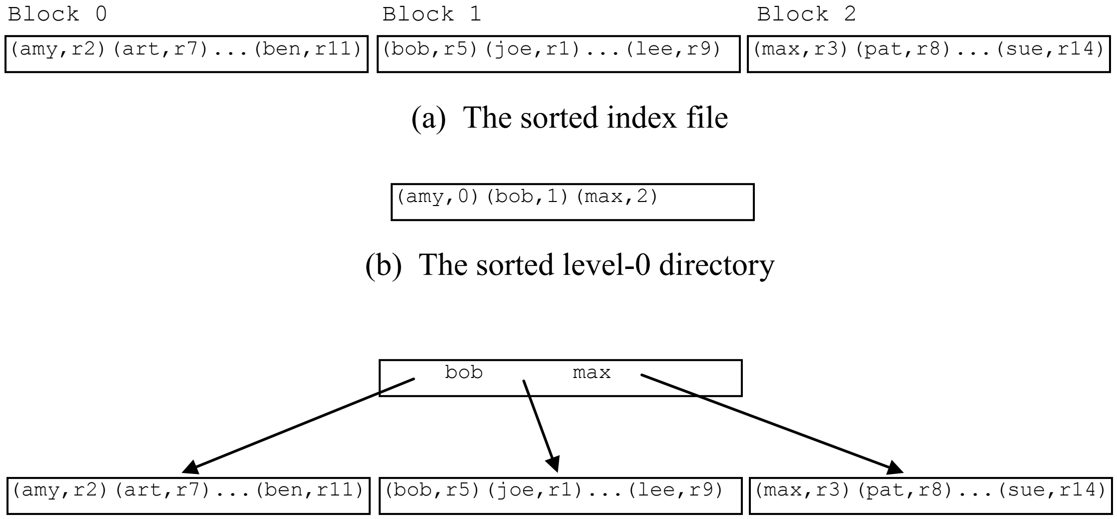

[a-bouquet][bourbon-couple][couplet-exclude] ...
[a-ability][abject-abscissa] ...
[bourbon-boycott][brace-branch] ...
...
[a: The highest grade]
[aback: Startled]
[abacus: A counting device]
...
[abject: Contemptible]
[abjure: Repudiate]
[able: Having skill]
...
...
[bourbon: A type of whiskey]
[bout: A contest]
[bovine: Resembling an ox]
...

Figure 21-11: Our improved dictionary, represented as a tree

21.4.2 The B-tree directory

The concept of a tree-structured directory can also be applied to sorted indexes. The index records will be stored in an index file. The level-0 directory will have a record for each block of the index file. These directory records will be of the form [dataval, block#], where the *dataval* is the dataval of the first index record in the block, and *block#* is the block number of that block.

For example, Figure 21-12(a) depicts the index file for a sorted index on the field *SName* of STUDENT. This index file consists of three blocks, with each block containing an unspecified number of records. Figure 21-12(b) depicts the level-0 directory for this index file. The directory consists of three records, one for each index block.


Block 0
Block 1
Block 2
(amy,r2)(art,r7)...(ben,r11)
(bob,r5)(joe,r1)...(lee,r9)
(max,r3)(pat,r8)...(sue,r14)
(amy,0)(bob,1)(max,2)
bob
max
(amy,r2)(art,r7)...(ben,r11)
(bob,r5)(joe,r1)...(lee,r9)
(max,r3)(pat,r8)...(sue,r14)

(a) The sorted index file

(b) The sorted level-0 directory

(c) The tree representation of the index and its directory

Figure 21-12: A B-tree index for *SName*

If the records in the directory are sorted by their dataval, then the range of values in each index block can be determined by comparing adjacent directory entries. For example, the three records in the directory of Figure 21-12(b) denote the following information:

- Block 0 of the index file contains index records whose datavals range from "amy" up to (but not including) "bob".
- Block 1 contains index records ranging from "bob" up to (but not including) "max".
- Block 2 contains index records ranging from "max" to the end.

In general, the dataval in the first directory record is not interesting, and is usually replaced by a special value (such as null), denoting "everything from the beginning".

A directory and its index blocks are usually represented graphically as a tree, as shown in Figure 21-12(c). That tree is an example of a *B-tree*[†]. Note how we can obtain the actual

[†] Historically, two slightly different versions of B-tree were developed. The version we are using is actually known as a *B^+-tree*, because it was developed second; the first version, which we won't consider,

directory records by pairing each arrow with the dataval preceding it. The dataval corresponding to the leftmost arrow is omitted in the tree representation, because it is not needed.

Given a dataval *v*, the directory can be used to find the index records having that dataval, or to insert a new index record for that dataval. The algorithms appear in Figure 21-13.

1. Search the directory block to find the directory record whose range of datavals contains *v*.
2. Read the index block pointed to by that directory record.
3. Examine the contents of this block to find the desired index records.

(a) Finding the index records having a specified dataval *v*

1. Search the directory block to find the directory record whose range of datavals contains *v*.
2. Read the index block pointed to by that directory record.
3. Insert the new index record into this block.

(b) Inserting a new index record having a specified dataval *v*

Figure 21-13: Algorithms to find and insert index records into the tree of Figure 21-12

We note two things about these algorithms. The first point is that steps 1 and 2 of these algorithms are identical. In other words, the insertion algorithm will insert an index record into the same block where the search algorithm will look for it – which is, of course, exactly what ought to happen. The second point is that each algorithm identifies a single index block where the desired records belong; thus, all index records having the same dataval must be in the same block.

The B-tree of Figure 21-12 is very simple, because the index is so small. As it gets larger, we will need to deal with the following three complications:

- The directory may be several blocks long.
- A newly-inserted index record may not fit into the block where it needs to go.
- There may be many index records having the same dataval.

These issues are addressed in the following subsections.

21.4.3 A directory tree

Suppose in our example that many more new employees have been inserted into the database, so that the index file now contains 8 blocks. If we assume (for sake of

preempted the *B-tree* designation. However, because the second version is by far more common in practice, we shall use the simpler (although slightly incorrect) term to denote it.

example) that at most 3 directory records fit into a block, then our B-tree directory will need at least 3 blocks. One idea might be to place these directory blocks into a file and scan them sequentially; however, such a scan would not be very efficient. A better idea corresponds to what we did with our improved dictionary: We need a "guide" to the level-0 directory.

That is, we now have two levels of directory blocks. Level 0 contains those blocks that point to the index blocks. Level 1 contains the block that points to the level 0 blocks. Pictorially, the B-tree might look like the tree of Figure 21-14. We can search this index by starting at the level-1 block. Suppose for example that the search key is "jim". The search key lies between "eli" and "lee", and so we follow the middle arrow, and search the level-0 block containing "joe". The search key is less than "joe", and so we follow the left arrow, and look in the index block containing "eli". All index records for "jim" (if any) will be in this block.

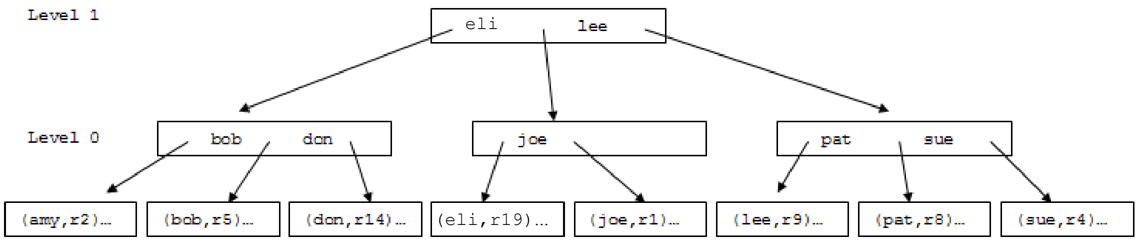


Figure 21-14: A B-tree having two directory levels

In general, whenever a level contains more than one directory block, we create directory blocks at the next higher level that point to them. Eventually the highest level will contain a single block. This block is called the *root* of the B-tree.

At this point, you should stop to make sure you are able to traverse a B-tree yourself. Using Figure 21-14, choose several names and make sure you can find the index block containing each name. There should be no ambiguity – given a dataval, there is exactly one index block where index records containing that dataval must be.

Also note the distribution of names in the directory records of the B-tree. For example, the value "eli" in the level-one node means that "eli" is the first name in the subtree pointed to by the middle arrow, which means that it is the first record of the first index block pointed to by the level-0 directory block. So even though "eli" does not appear explicitly in the level-0 block, it manages to make an appearance in the level-1 block. In fact, it turns out that the first dataval of each index block (except the very first block) appears exactly once in some directory block at some level of the B-tree.

A search of a B-tree requires accessing one directory block at each level, plus one index block. Thus the search cost is equal to the number of directory levels plus 1. To see the practical impact of this formula, consider the example at the end of Section 21.2.1, where we calculated the search costs for a static hash index on *SName*, using 4K-byte blocks. As before, each index record will be 22 bytes, with 178 index records fitting into a block. Each directory record is 18 bytes (14 bytes for the dataval and 4 bytes for the block number), so 227 directory records will fit in a block. Thus:

- A 0-level B-tree, which can be searched using 2 disk accesses, can hold up to 227*178 = 40,406 index records.
- A 1-level B-tree, which can be searched using 3 disk accesses, can hold up to 227*227*178 = 9,172,162 index records.
- A 2-level B-tree, which can be searched using 4 disk accesses, can hold up to 227*227*227*178 = 2,082,080,774 index records.

In other words, B-tree indexes are exceptionally efficient. Any desired data record can be retrieved in no more than 5 disk accesses, unless its table is unusually large[†]. If a commercial database system implements only one indexing strategy, it almost certainly uses a B-tree.

21.4.4 Inserting records

If we want to insert a new index record, then the algorithm of Figure 21-13(b) implies that there is exactly one index block where it can be inserted. What do we do if that block has no more room? As with extendable hashing, the solution is to *split* the block.

Splitting an index block entails the following activities:

- We allocate a new block in the index file.
- We move the high-valued half of the index records into this new block.
- We create a directory record for the new block.
- We insert this new directory record into the same level-0 directory block that pointed to the original index block.

For example, consider Figure 21-14, and suppose all index blocks are full. To insert the new index record (hal, r_{55}), we follow the B-tree directory and determine that the record belongs in the index block that contains "eli". We therefore split this block, moving the upper half of its records into the new block. Suppose the new block is block 8 of the index file, and its first record is (jim, r_{48}). We then insert the directory record (jim, 8) into the level-0 directory block. The resulting subtree appears in Figure 21-15.

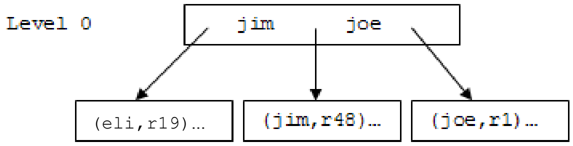


Figure 21-15: The effect of splitting an index block

[†] And if we consider buffering, things look even better. If the index is used often, then the root block and many of the blocks in the level below it will probably already be in buffers, so it is likely that even fewer disk accesses will be required.

In this case, there was room in the level-0 block for the new directory record. If there is no room, then that directory block also needs to split. For example, return to Figure 21-14 and suppose that an index record (zoe, r_{56}) is inserted. This record will cause the rightmost index block to split – Suppose that the new block is number 9 and its first dataval is “tom”. Then (tom, 9) is inserted into the rightmost level-0 directory block. However, there is no room in that level-0 block, and so it also splits. Two directory records stay in the original block, and two move to the new block (which is, say, block 4 of the directory file). The resulting directory and index blocks appear in Figure 21-16. Note that the directory record for “sue” still exists, but is not visible in the picture because it is the first record of its block.

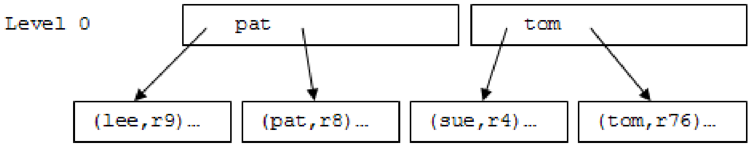


Figure 21-16: Splitting a directory block

We are not done. The new level-0 block requires inserting a record into a level-1 directory block, and so the same record-insertion process happens recursively. The new directory record to be inserted is (sue, 4). The value “sue” is used because it is the smallest dataval in the subtree of the new directory block. This recursive insertion of directory records continues up the B-tree. If the root block splits, then a new root block is created, and the B-tree gains an additional level. This is exactly what happens in our example. The level-1 block has no room and so it also splits, creating a new level-1 block, and a new level-2 block to be the root. The resulting B-tree appears in Figure 21-17.

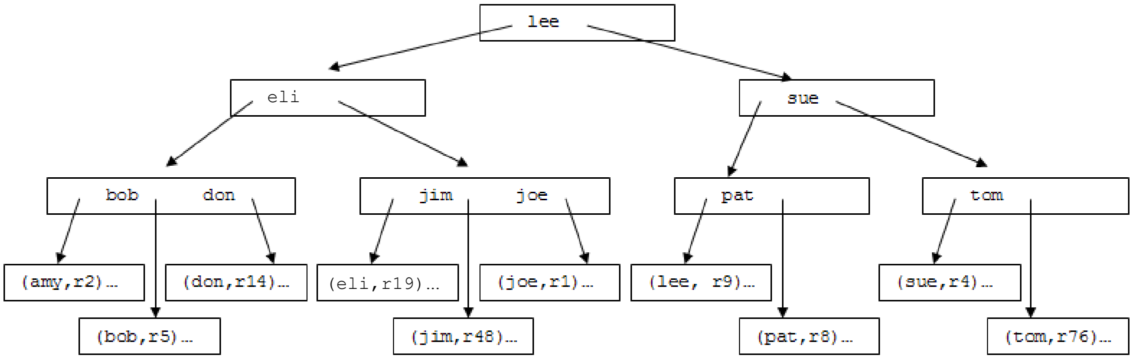


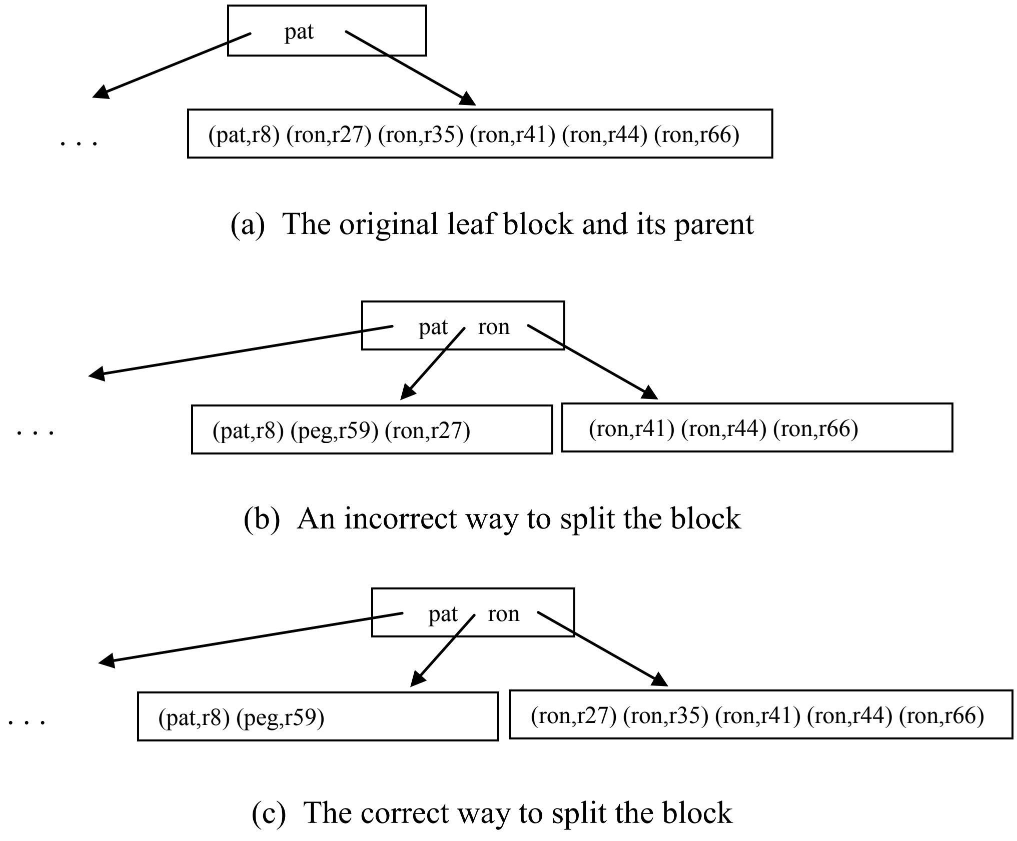
Figure 21-17: Splitting the root of the B-tree

Note that splitting a block turns a full block into two half-full blocks. In general, the capacity of a B-tree will range from 50% to 100%.

21.4.5 Duplicate datavals

Recall from Chapter 6 that an index is useful only when it is selective. That is, if too many index records have the same dataval, then it is better to ignore the index and just scan the data file directly. So although it is possible for an index to contain an arbitrary number of records having the same dataval, in practice there will not be that many, and most likely not enough to fill multiple blocks. Nevertheless, a B-tree must be able to handle such cases.

To see what the issues are, consider Figure 21-17 again, and suppose that there are several records for the dataval "ron". Note that all of those records must be in the same leaf block of the B-tree – namely, the block that contains "pat". Suppose that the actual contents of that block is shown in Figure 21-18(a). Suppose now that we insert a record for "peg", and this record causes the block to split. Figure 21-18(b) shows the result of splitting the block evenly: The records for "ron" wind up in different blocks.


pat
. . .
(pat,r8) (ron,r27) (ron,r35) (ron,r41) (ron,r44) (ron,r66)

(a) The original leaf block and its parent

pat ron
. . .
(pat,r8) (peg,r59) (ron,r27)
(ron,r41) (ron,r44) (ron,r66)

(b) An incorrect way to split the block

pat ron
. . .
(pat,r8) (peg,r59)
(ron,r27) (ron,r35) (ron,r41) (ron,r44) (ron,r66)

(c) The correct way to split the block

Figure 21-18: Splitting a leaf block that has duplicate values

The B-tree of Figure 21-18(b) is clearly unacceptable, because the record for Ron that remained in Pat's block is not accessible. We have the following rule:

> *When you split a block, you must place*
> *all records having the same dataval in the same block.*

This rule is just common sense. When you use the B-tree directory to look for index records having a particular search key, the directory will always point you to a single leaf block. Consequently, if index records with that search key exist in other blocks, then those records will never be found.

The consequence of this rule is that we may not be able to split an index block evenly. Figure 21-18(c) depicts the only reasonable way to perform the split, by placing the four "ron" records in the new block.

An index block can always be split if its records contain at least two different datavals. The only real problem occurs when all of the records in the index block have the same dataval. In this case, splitting is of no use. Instead, the best approach is to use an overflow block.

For example, suppose that we start from Figure 21-18(c) and insert records for several more students named Ron. Instead of splitting the block, we create a new leaf block and move all but one of the "ron" records into it. This new block is the overflow block. The old block links to the overflow block, as shown in Figure 21-19.

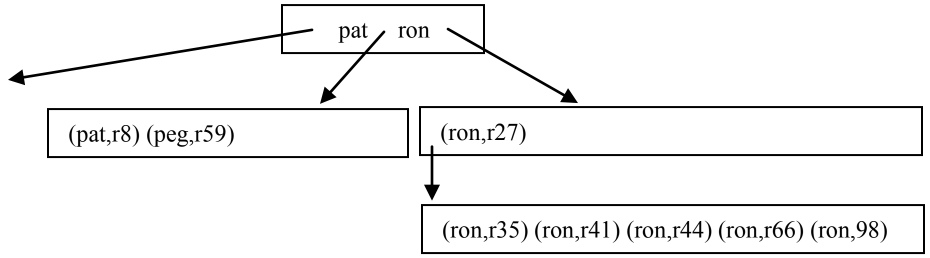


Figure 21-29: Using an overflow chain to store records having the same dataval

Note how the old block has been nearly emptied, which allows us to insert additional records into it (which may or may not be for students named "ron"). If the block fills up again, there are two possibilities.

- If there are at least two different datavals in the block, then it splits.
- If the block contains only "ron" records, then another overflow block is created and chained to the existing one.

In general, a leaf block can contain a chain of overflow blocks. Each overflow block will be completely filled. All of the records in the overflow chain will always have the same dataval, which will always be the same as the dataval of the first record in the non-overflow block.

Suppose that we are searching for index records having a particular search key. We follow the B-tree directory to a particular leaf block. If the search key is not the first key of the block, then we examine the records in the block as before. If the search key is the first one, then we also need to use the records in the overflow chain, if one exists.

Although the index records in the B-tree may contain duplicate datavals, the directory entries will not. The reason is that the only way to get a dataval into a directory entry is to split a leaf block; the first dataval of the new block is added to the directory. But the dataval that is first in its block will never get split again – if the block fills with records having that dataval, then an overflow block will be created instead.

21.4.6 Implementing B-Tree Pages

The SimpleDB code to implement B-trees lives in the package *simpledb.index.btree*. This package contains four principal classes: *BTreeIndex*, *BTreeDir*, *BTreeLeaf*, and *BTreePage*. Classes *BTreeDir* and *BTreeLeaf* implement the directory and index blocks of a B-tree, respectively[†]. Although the directory and leaf blocks contain different kinds of records and are used in different ways, they have common requirements, such as the need to insert entries in sorted order and to split themselves. The class *BTreePage* contains this common code. The class *BTreeIndex* implements the actual B-tree operations, as specified by the *Index* interface.

We begin by examining the class *BTreePage.* Records in a B-tree page have the following requirements:

- The records need to be maintained in sorted order.
- The records do not need to have a permanent id, which means that they can be moved around within the page as needed.
- A page needs to be able to split its records with another page.
- Each page needs an integer to serve as a flag. (A directory page uses the flag to hold its level, and a leaf page uses the flag to point to its overflow block.)

That is, you can think of a B-tree page as holding a sorted list of records (as opposed to a record page, which conceptually holds an unsorted array of records). When a new record is inserted into the page, its position in the sort order is determined and the records following it are shifted one place to the right to make room. Similarly when a record is deleted, the records following it are shifted to the left to fill in the hole. In order to implement this list-like behavior, the page must also store an integer that holds the current number of records in the page.

The code for class *BTreePage* appears in Figure 21-20.

[†] We use the term *leaf* to denote index blocks, because they form the leaves of the B-tree. The SimpleDB implementation uses "leaf" to avoid confusion with the *BTreeIndex* class, which implements the *Index* interface.

```
public class BTreePage {
   private Block currentblk;
   private TableInfo ti;
   private Transaction tx;
   private int slotsize;

   public BTreePage(Block currentblk, TableInfo ti, Transaction tx) {
      this.currentblk = currentblk;
      this.ti = ti;
      this.tx = tx;
      slotsize = ti.recordLength();
      tx.pin(currentblk);
   }

   public int findSlotBefore(Constant searchkey) {
      int slot = 0;
      while (slot < getNumRecs() &&
             getDataVal(slot).compareTo(searchkey) < 0)
         slot++;
      return slot-1;
   }

   public void close() {
      if (currentblk != null)
         tx.unpin(currentblk);
      currentblk = null;
   }

   public boolean isFull() {
      return slotpos(getNumRecs()+1) >= BLOCK_SIZE;
   }

   public Block split(int splitpos, int flag) {
      Block newblk = appendNew(flag);
      BTreePage newpage = new BTreePage(newblk, ti, tx);
      transferRecs(splitpos, newpage);
      newpage.setFlag(flag);
      newpage.close();
      return newblk;
   }

   public Constant getDataVal(int slot) {
      return getVal(slot, "dataval");
   }

   public int getFlag() {
      return tx.getInt(currentblk, 0);
   }

   public void setFlag(int val) {
      tx.setInt(currentblk, 0, val);
   }

   public Block appendNew(int flag) {
      return tx.append(ti.fileName(), new BTPageFormatter(ti, flag));
   }
   // Methods called only by BTreeDir

   public int getChildNum(int slot) {
      return getInt(slot, "block");
   }

   public void insertDir(int slot, Constant val, int blknum) {
      insert(slot);
      setVal(slot, "dataval", val);
      setInt(slot, "block", blknum);
   }

   // Methods called only by BTreeLeaf

   public RID getDataRid(int slot) {
      return new RID(getInt(slot, "block"), getInt(slot, "id"));
   }

   public void insertLeaf(int slot, Constant val, RID rid) {
      insert(slot);
      setVal(slot, "dataval", val);
      setInt(slot, "block", rid.blockNumber());
      setInt(slot, "id", rid.id());
   }

   public void delete(int slot) {
      for (int i=slot+1; i<getNumRecs(); i++)
         copyRecord(i, i-1);
      setNumRecs(getNumRecs()-1);
      return;
   }

   public int getNumRecs() {
      return tx.getInt(currentblk, INT_SIZE);
   }

   // Private methods

   private int getInt(int slot, String fldname) {
      int pos = fldpos(slot, fldname);
      return tx.getInt(currentblk, pos);
   }

   private String getString(int slot, String fldname) {
      int pos = fldpos(slot, fldname);
      return tx.getString(currentblk, pos);
   }

   private Constant getVal(int slot, String fldname) {
      int type = ti.schema().type(fldname);
      if (type == INTEGER)
         return new IntConstant(getInt(slot, fldname));
      else
         return new StringConstant(getString(slot, fldname));
   }

   private void setInt(int slot, String fldname, int val) {
      int pos = fldpos(slot, fldname);
      tx.setInt(currentblk, pos, val);
   }

   private void setString(int slot, String fldname, String val) {
      int pos = fldpos(slot, fldname);
      tx.setString(currentblk, pos, val);
   }

   private void setVal(int slot, String fldname, Constant val) {
      int type = ti.schema().type(fldname);
      if (type == INTEGER)
         setInt(slot, fldname, (Integer)val.asJavaVal());
      else
         setString(slot, fldname, (String)val.asJavaVal());
   }

   private void setNumRecs(int n) {
      tx.setInt(currentblk, INT_SIZE, n);
   }

   private void insert(int slot) {
      for (int i=getNumRecs(); i>slot; i--)
         copyRecord(i-1, i);
      setNumRecs(getNumRecs()+1);
   }

   private void copyRecord(int from, int to) {
      Schema sch = ti.schema();
      for (String fldname : sch.fields())
         setVal(to, fldname, getVal(from, fldname));
   }

   private void transferRecs(int slot, BTreePage dest) {
      int destslot = 0;
      while (slot < getNumRecs()) {
         dest.insert(destslot);
         Schema sch = ti.schema();
         for (String fldname : sch.fields())
            dest.setVal(destslot, fldname, getVal(slot, fldname));
         delete(slot);
         destslot++;
      }
   }

   private int fldpos(int slot, String fldname) {
      int offset = ti.offset(fldname);
      return slotpos(slot) + offset;
   }

   private int slotpos(int slot) {
      return INT_SIZE + INT_SIZE + (slot * slotsize);
   }
}
```

Figure 21-20: The code for the SimpleDB class *BTreePage*

The most interesting method in this class is *findSlotBefore*. This method takes a search key k as argument finds the smallest slot x such that $k \leq dataval(x)$; it then returns the slot before that. The reason for this behavior is that it accommodates all of the ways that pages can be searched. For example, it acts like a *beforeFirst* operation on leaf pages, so that a call to *next* will retrieve the first record having that search key.

We now turn to the leaf blocks of the B-tree and the class *BTreeLeaf*; its code appears in Figure 21-21.

```
public class BTreeLeaf {
   private TableInfo ti;
   private Transaction tx;
   private String filename;
   private Constant searchkey;
   private BTreePage contents;
   private int currentslot;

   public BTreeLeaf(Block blk, TableInfo ti, Constant searchkey,
                                           Transaction tx) {
      this.ti = ti;
      this.tx = tx;
      this.searchkey = searchkey;
      filename = blk.fileName();
      contents = new BTreePage(blk, ti, tx);
      currentslot = contents.findSlotBefore(searchkey);
   }

   public void close() {
      contents.close();
   }


   public boolean next() {
      currentslot++;
      if (currentslot >= contents.getNumRecs())
         return tryOverflow();
      else if (contents.getDataVal(currentslot).equals(searchkey))
         return true;
      else
         return tryOverflow();
   }

   public RID getDataRid() {
      return contents.getDataRid(currentslot);
   }

   public void delete(RID datarid) {
      while(next())
         if(getDataRid().equals(datarid)) {
            contents.delete(currentslot);
            return;
         }
   }
   public DirEntry insert(RID datarid) {
      currentslot++;
      contents.insertLeaf(currentslot, searchkey, datarid);
      if (!contents.isFull())
         return null;
      // else page is full, so split it
      Constant firstkey = contents.getDataVal(0);
      Constant lastkey  = contents.getDataVal(contents.getNumRecs()-1);
      if (lastkey.equals(firstkey)) {
         // create an overflow block, using splitpos=1
         Block newblk = contents.split(1, contents.getFlag());
         contents.setFlag(newblk.number());
         return null;
      }
      else {
         int splitpos = contents.getNumRecs() / 2;
         Constant splitkey = contents.getDataVal(splitpos);
         if (splitkey.equals(firstkey)) {
            // move right, looking for the next key
            while (contents.getDataVal(splitpos).equals(splitkey))
               splitpos++;
            splitkey = contents.getDataVal(splitpos);
         }
         else {
            // move left, looking for first entry having that key
            while (contents.getDataVal(splitpos-1).equals(splitkey))
               splitpos--;
         }
         Block newblk = contents.split(splitpos, -1);
         return new DirEntry(splitkey, newblk.number());
      }
   }

   private boolean tryOverflow() {
      Constant firstkey = contents.getDataVal(0);
      int flag = contents.getFlag();
      if (!searchkey.equals(firstkey) || flag < 0)
         return false;
      contents.close();
      Block nextblk = new Block(filename, flag);
      contents = new BTreePage(nextblk, ti, tx);
      currentslot = 0;
      return true;
   }
}
```

Figure 21-21: The code for the SimpleDB class *BTreeLeaf*

The constructor first creates a B-tree page for the specified block, and then calls *findSlotBefore* to position itself immediately before the first record containing the search key. A call to *next* moves to the next record, and returns *true* or *false* depending on whether that record has the desired search key. The call to *tryOverflow* handles the possibility that the leaf block contains an overflow chain.

The methods *delete* and *insert* assume that the current slot of the page has already been set by a call to *findSlotBefore*. Method *delete* repeatedly calls *next* until it encounters the index record having the specified rid, and deletes that record. Method *insert* moves to the next record, which means that it is now positioned at the first record greater than or equal to that search key. The new record is inserted in that spot. Note that if the page already contains records having that search key, then the new record will be inserted at the front of the list. Method *insert* returns an object of type *DirEntry* (that is, a directory record). If the insertion does not cause the block to split, then this return value is null. If a split occurs, then the return value is the (dataval, blocknumber) entry corresponding to the new index block.

The class *BTreeDir* implements directory blocks; its code appears in Figure 21-22.

```
public class BTreeDir {
   private TableInfo md;
   private Transaction tx;
   private String filename;
   private BTreePage contents;

   BTreeDir(Block blk, TableInfo md, Transaction tx) {
      this.md = md;
      this.tx = tx;
      filename = blk.fileName();
      contents = new BTreePage(blk, md, tx);
   }

   public void close() {
      contents.close();
   }

   public int search(Constant searchkey) {
      Block childblk = findChildBlock(searchkey);
      while (contents.getFlag() > 0) {
         contents.close();
         contents = new BTreePage(childblk, md, tx);
         childblk = findChildBlock(searchkey);
      }
      return childblk.number();
   }

   public void makeNewRoot(DirEntry e) {
      Constant firstval = contents.getDataVal(0);
      int level = contents.getFlag();
      Block newblk = contents.split(0, level);
      DirEntry oldroot = new DirEntry(firstval, newblk.number());
      insertEntry(oldroot);
      insertEntry(e);
      contents.setFlag(level+1);
   }

   public DirEntry insert(DirEntry e) {
      if (contents.getFlag() == 0)
         return insertEntry(e);
      Block childblk = findChildBlock(e.dataVal());
      BTreeDir child = new BTreeDir(childblk, md, tx);
      DirEntry myentry = child.insert(e);
      child.close();
      return (myentry != null) ? insertEntry(myentry) : null;
   }

   private DirEntry insertEntry(DirEntry e) {
      int newslot = 1 + contents.findSlotBefore(e.dataVal());
      contents.insertDir(newslot, e.dataVal(), e.blockNumber());
      if (!contents.isFull())
         return null;
      // else page is full, so split it
      int level = contents.getFlag();
      int splitpos = contents.getNumRecs() / 2;
      Constant splitval = contents.getDataVal(splitpos);
      Block newblk = contents.split(splitpos, level);
      return new DirEntry(splitval, newblk.number());
   }

   private Block findChildBlock(Constant searchkey) {
      int slot = contents.findSlotBefore(searchkey);
      if (contents.getDataVal(slot+1).equals(searchkey))
         slot++;
      int blknum = contents.getChildNum(slot);
      return new Block(filename, blknum);
   }
}
```

Figure 21-22: The code for the SimpleDB class *BTreeDir*

Methods *search* and *insert* both start at the root, moving down the tree until the level-0 directory block associated with the search key is located. Method *search* uses a simple while-loop to move down the tree; when the level-0 block is found, it searches that page and returns the block number of the leaf containing the search key. Method *insert* uses recursion to move down the tree. The return value of the recursive call indicates whether the insertion caused its child page to split; it so, then the method *insertEntry* is called to insert a new directory record into the page. If this insertion causes the page to split, the directory record for the new page is passed back to the page's parent. A null value indicates that no split occurred.

The method *makeNewRoot* is invoked when a call to *insert* on the root page returns a non-null value. Since the root must always be at block 0 of the directory file, this method allocates a new block, copies the contents of block 0 to the new block, and initializes block 0 as the new root. The new root will always have two entries: The first entry will refer to the old root; the second entry will refer to the newly-split block (that was passed in as an argument to *makeNewRoot*).

21.4.7 Implementing the B-Tree Index

Now that we have examined how B-tree pages are implemented, we finally turn to how they are used. The class *BTreeIndex* implements the methods of the *Index* interface, coordinating the use of directory and leaf pages; see Figure 21-23. Its constructor does most of the heavy lifting. From the supplied *IndexInfo* object, it retrieves the table info of the leaf blocks. It then constructs the schema of the directory blocks by extracting the corresponding information from the leaf schema, and from there constructs the table info of the directory blocks. Finally, it formats the root if necessary, inserting an entry that points to block 0 of the leaf file.

```
public class BTreeIndex implements Index {
   private Transaction tx;
   private TableInfo dirTi, leafTi;
   private BTreeLeaf leaf = null;
   private Block rootblk;

   public BTreeIndex(String idxname, Schema leafsch, Transaction tx) {
      this.tx = tx;
      // deal with the leaves
      String leaftbl = idxname + "leaf";
      leafTi = new TableInfo(leaftbl, leafsch);
      if (tx.size(leafTi.fileName()) == 0)
         tx.append(leafTi.fileName(), new BTPageFormatter(leafTi, -1));

      // deal with the directory
      Schema dirsch = new Schema();
      dirsch.add("block",   leafsch);
      dirsch.add("dataval", leafsch);
      String dirtbl = idxname + "dir";
      dirTi = new TableInfo(dirtbl, dirsch);
      rootblk = new Block(dirTi.fileName(), 0);
      if (tx.size(dirTi.fileName()) == 0) { // format new dir file
         tx.append(dirTi.fileName(), new BTPageFormatter(dirTi, 0));
         BTreePage page = new BTreePage(rootblk, dirTi, tx);
         int fldtype = dirsch.type("dataval");
         Constant minval = (fldtype == INTEGER) ?
                           new IntConstant(Integer.MIN_VALUE) :
                           new StringConstant("");
         page.insertDir(0, minval, 0);
         page.close();
      }
   }

   public void beforeFirst(Constant searchkey) {
      close();
      BTreeDir root = new BTreeDir(rootblk, dirTi, tx);
      int blknum = root.search(searchkey);
      root.close();
      Block leafblk = new Block(leafTi.fileName(), blknum);
      leaf = new BTreeLeaf(leafblk, leafTi, searchkey, tx);
   }

   public boolean next() {
      return leaf.next();
   }

   public RID getDataRid() {
      return leaf.getDataRid();
   }

   public void insert(Constant dataval, RID datarid) {
      beforeFirst(dataval);
      DirEntry e = leaf.insert(datarid);
      leaf.close();
      if (e == null)
         return;
      BTreeDir root = new BTreeDir(rootblk, dirTi, tx);
      DirEntry e2 = root.insert(e);
      if (e2 != null)
         root.makeNewRoot(e2);
      root.close();
   }

   public void delete(Constant dataval, RID datarid) {
      beforeFirst(dataval);
      leaf.delete(datarid);
      leaf.close();
   }

   public void close() {
      if (leaf != null)
         leaf.close();
   }

   public static int searchCost(int numblocks, int rpb) {
      return 1 + (int)(Math.log(numblocks) / Math.log(rpb));
   }
}
```

Figure 21-23: The code for the SimpleDB class *BTreeIndex*

Each *BTreeIndex* object holds an open *BTreeLeaf* object. This leaf object keeps track of the current index record: it is initialized by a call to method *beforeFirst*, incremented by a call to *next*, and accessed by calls to *getDataRid*, *insert*, and *delete*. The method *beforeFirst* initializes this leaf object by calling method *search* from the root directory page. Note that once the leaf page has been located, the directory is no longer needed, and its pages can be closed.

Method *insert* has two parts. The first part locates the appropriate leaf page and inserts the index record into it. If the leaf page splits, then the method inserts the index record for the new leaf into the directory, starting the recursion at the root. A non-null return value from the root means that the root has split, and so *makeNewRoot* is called.

Method *delete* deletes the index record from the leaf, but does not try to modify the directory. Another strategy would be to perform the deletion through the B-tree, as with insertions. Such as strategy would allow the directory blocks to coalesce if they became sufficiently empty. However, the algorithm for coalescing blocks is complex and error-prone, and is rarely implemented. The reason is that databases rarely get smaller – deletions are usually followed by other insertions. Consequently, it makes sense to leave the nearly-empty directory blocks in place, assuming that records will soon be inserted into them.

21.5 Index-Aware Operator Implementations

Now that we have seen how indexes can be implemented, we turn our attention to the issue of how the query planner can take advantage of them. Given an SQL query, the planner has two tasks to perform: It must determine the appropriate query tree, and it must choose a plan for each operator in the tree. This second task is trivial for the basic planner of Chapter 19, because it only knows about one implementation for each operator. For example, it always implements a select node using a *SelectPlan*, regardless of whether an index is available or appropriate.

In order to for the planner to be able to construct a plan that uses an index, it needs to have operator implementations that use indexes. In this section we develop such implementations for the *select* and *join* operators. Given a query, the planner is then free to incorporate these implementations in its plan.

Of course, the planning process becomes much more complicated when relational operators can have more than one implementation. The planner must be able to consider multiple plans for a query, some which use indexes, and some that do not; it then must decide which plan is the most efficient. This feature is called *query optimization*, and is the topic of Chapter 24.

21.5.1 An indexed implementation of *select*

The SimpleDB class *IndexSelectPlan* implements the *select* operator. Its code appears in Figure 21-24. The constructor takes four arguments: The plan for the underlying table, which is assumed to be a *TablePlan*; the information about the applicable index; the selection constant; and the current transaction. The method *open* opens the index and passes it (and the constant) to the *IndexSelectScan* constructor. The methods *blocksAccessed*, *recordsOutput*, and *distinctValues* implement cost-estimation formulas, using methods provided by the *IndexInfo* class.

```
public class IndexSelectPlan implements Plan {
   private Plan p;
   private IndexInfo ii;
   private Constant val;

   public IndexSelectPlan(Plan p, IndexInfo ii, Constant val,
                                                Transaction tx) {
      this.p = p;
      this.ii = ii;
      this.val = val;
   }

   public Scan open() {
      TableScan ts = (TableScan) p.open();
      Index idx = ii.open();
      return new IndexSelectScan(idx, val, ts);
   }

   public int blocksAccessed() {
      return ii.blocksAccessed() + recordsOutput();
   }

   public int recordsOutput() {
      return ii.recordsOutput();
   }

   public int distinctValues(String fldname) {
      return ii.distinctValues(fldname);
   }

   public Schema schema() {
      return p.schema();
   }
}
```

Figure 21-24: The code for the SimpleDB class *IndexSelectPlan*

The code for *IndexSelectScan* appears in Figure 21-25. The *Index* variable *idx* holds the current index record, and the *TableScan* variable *ts* holds the current data record. The call to *next* moves to the next index record having the specified search constant; the tablescan is then positioned at the data record having the datarid value of the current index record. Note that the table scan is never scanned; its current record is always obtained via the datarid of an index record.

```
public class IndexSelectScan implements Scan {
   private Index idx;
   private Constant val;
   private TableScan ts;

   public IndexSelectScan(Index idx, Constant val, TableScan ts) {
      this.idx = idx;
      this.val = val;
      this.ts  = ts;
      beforeFirst();
   }

   public void beforeFirst() {
      idx.beforeFirst(val);
   }

   public boolean next() {
      boolean ok = idx.next();
      if (ok) {
         RID rid = idx.getDataRid();
         ts.moveToRid(rid);
      }
      return ok;
   }

   public void close() {
      idx.close();
      ts.close();
   }

   public Constant getVal(String fldname) {
      return ts.getVal(fldname);
   }

   public int getInt(String fldname) {
      return ts.getInt(fldname);
   }

   public String getString(String fldname) {
      return ts.getString(fldname);
   }

   public boolean hasField(String fldname) {
      return ts.hasField(fldname);
   }
}
```

Figure 21-25: The code for the SimpleDB class *IndexSelectScan*

The remaining scan methods (*getVal*, *getInt*, etc.) pertain to the current data record, and thus are obtained directly from the table scan.

21.5.2 An indexed implementation of *join*

An *index join* is an implementation of the *join* operator. Consider the operation

```
join(T1, T2, p)
```

The index join implementation applies when the following conditions hold:

- The underlying table T_2 must be stored.
- The predicate *p* must be of the form "*A*=*B*", where *A* is a field from T_1 and *B* is a field from T_2. (That is, *p* must be an equijoin.)
- Table T_2 has an index on *B*.

In such a case, the join can be implemented similarly to the algorithm of Figure 6-13. In particular, we iterate through T_1. For each T_1 record, we grab its *A*-value and use it to search the index. We then use the datarid of each matching index record to lookup the corresponding record from T_2. These records are exactly the records that match the T_1 record.

The classes *IndexJoinPlan* and *IndexJoinScan* implement index joins; their code appears in Figures 21-26 and 21-27. An index join is implemented similarly to a product; the difference is that instead of repeatedly scanning the inner table, the code only has to repeatedly search the index.

```
public class IndexJoinPlan implements Plan {
   private Plan p1, p2;
   private IndexInfo ii;
   private String joinfield;
   private Schema sch = new Schema();

   public IndexJoinPlan(Plan p1, Plan p2, IndexInfo ii,
                        String joinfield, Transaction tx) {
      this.p1 = p1;
      this.p2 = p2;
      this.ii = ii;
      this.joinfield = joinfield;
      sch.addAll(p1.schema());
      sch.addAll(p2.schema());
   }

   public Scan open() {
      Scan s = p1.open();
      // throws an exception if p2 is not a table plan
      TableScan ts = (TableScan) p2.open();
      Index idx = ii.open();
      return new IndexJoinScan(s, idx, joinfield, ts);
   }

   public int blocksAccessed() {
      return p1.blocksAccessed()
          + (p1.recordsOutput() * ii.blocksAccessed())
          + recordsOutput();
   }

   public int recordsOutput() {
      return p1.recordsOutput() * ii.recordsOutput();
   }

   public int distinctValues(String fldname) {
      if (p1.schema().hasField(fldname))
         return p1.distinctValues(fldname);
      else
         return p2.distinctValues(fldname);
   }

   public Schema schema() {
      return sch;
   }
}
```

Figure 21-26: The code for the SimpleDB class *IndexJoinPlan*

```
public class IndexJoinScan implements Scan {
   private Scan s;
   private TableScan ts;  // the data table
   private Index idx;
   private String joinfield;

   public IndexJoinScan(Scan s, Index idx, String joinfield,
                                           TableScan ts) {
      this.s = s;
      this.idx  = idx;
      this.joinfield = joinfield;
      this.ts = ts;
      beforeFirst();
   }

   public void beforeFirst() {
      s.beforeFirst();
      s.next();
      resetIndex();
   }

   public boolean next() {
      while (true) {
         if (idx.next()) {
            ts.moveToRid(idx.getDataRid());
            return true;
         }
         if (!s.next())
            return false;
         resetIndex();
      }
   }

   public void close() {
      s.close();
      idx.close();
      ts.close();
   }

   public Constant getVal(String fldname) {
      if (ts.hasField(fldname))
         return ts.getVal(fldname);
      else
         return s.getVal(fldname);
   }

   public int getInt(String fldname) {
      if (ts.hasField(fldname))
         return ts.getInt(fldname);
      else
         return s.getInt(fldname);
   }

   public String getString(String fldname) {
      if (ts.hasField(fldname))
         return ts.getString(fldname);
      else
```

```
         return s.getString(fldname);
   }

   public boolean hasField(String fldname) {
      return ts.hasField(fldname) || s.hasField(fldname);
   }

   private void resetIndex() {
      Constant searchkey = s.getVal(joinfield);
      idx.beforeFirst(searchkey);
   }
}
```

Figure 21-27: The code for the SimpleDB class *IndexJoinScan*

21.6 Index Update Planning

If a database system supports indexing, its planner must ensure that whenever a data record is updated, there is a corresponding change to each of its index records. The code fragment of Figure 21-3 showed the kind of code that the planner needs to execute. We now see how the planner does it.

Package *simpledb.index.planner* contains the planner class *IndexUpdatePlanner*, which modifies the basic update planner; its code appears in Figure 21-28.

```
public class IndexUpdatePlanner implements UpdatePlanner {

   public int executeInsert(InsertData data, Transaction tx) {
      Plan plan = new TablePlan(data.tableName(), tx);

      // first, insert the record
      UpdateScan s = (UpdateScan) plan.open();
      s.insert();
      RID rid = s.getRid();

      // then modify each field; insert an index record if appropriate
      Map<String,IndexInfo> indexes =
                  SimpleDB.mdMgr().getIndexInfo(data.tableName(), tx);
      Iterator<Constant> valIter = data.vals().iterator();
      for (String fldname : data.fields()) {
         Constant val = valIter.next();
         s.setVal(fldname, val);

         IndexInfo ii = indexes.get(fldname);
         if (ii != null) {
            Index idx = ii.open();
            idx.insert(val, rid);
            idx.close();
         }
      }
      s.close();
      return 1;
   }

   public int executeDelete(DeleteData data, Transaction tx) {
      String tblname = data.tableName();
      Plan p = new TablePlan(tblname, tx);
      p = new SelectPlan(p, data.pred());
      Map<String,IndexInfo> indexes =
                            SimpleDB.mdMgr().getIndexInfo(tblname, tx);

      UpdateScan s = (UpdateScan) p.open();
      int count = 0;
      while(s.next()) {
         // first, delete the record's RID from every index
         RID rid = s.getRid();
         for (String fldname : indexes.keySet()) {
            Constant val = s.getVal(fldname);
            Index idx = indexes.get(fldname).open();
            idx.delete(val, rid);
            idx.close();
         }
         // then delete the record
         s.delete();
         count++;
      }
      s.close();
      return count;
   }

   public int executeModify(ModifyData data, Transaction tx) {
      String tblname = data.tableName();
      String fldname = data.targetField();
      Plan p = new TablePlan(tblname, tx);
      p = new SelectPlan(p, data.pred());

      IndexInfo ii =
               SimpleDB.mdMgr().getIndexInfo(tblname, tx).get(fldname);
      Index idx = (ii == null) ? null : ii.open();

      UpdateScan s = (UpdateScan) p.open();
      int count = 0;
      while(s.next()) {
         // first, update the record
         Constant newval = data.newValue().evaluate(s);
         Constant oldval = s.getVal(fldname);
         s.setVal(data.targetField(), newval);

         // then update the appropriate index, if it exists
         if (idx != null) {
            RID rid = s.getRid();
            idx.delete(oldval, rid);
            idx.insert(newval, rid);
         }
         count++;
      }
      if (idx != null) idx.close();
      s.close();
      return count;
   }

   public int executeCreateTable(CreateTableData data, Transaction tx){
      SimpleDB.mdMgr().createTable(data.tableName(), data.newSchema(),
                                   tx);
      return 0;
   }

   public int executeCreateView(CreateViewData data, Transaction tx) {
      SimpleDB.mdMgr().createView(data.viewName(), data.viewDef(), tx);
      return 0;
   }

   public int executeCreateIndex(CreateIndexData data, Transaction tx){
      SimpleDB.mdMgr().createIndex(data.indexName(), data.tableName(),
                                   data.fieldName(), tx);
      return 0;
   }
}
```

Figure 21-28: The code for the SimpleDB class *IndexUpdatePlanner*

The method *executeInsert* retrieves the index information of the mentioned table. As in the basic planner, the method calls *setVal* to set the initial value of each specified field. After each call to *setVal*, the planner looks to see if there is an index on that field; if there is, then it inserts a new record into that index.

The method *executeDelete* constructs a scan of records to be deleted, as in the basic planner. Before each of these data records is deleted, the method uses the record's field values to determine which index records need to be deleted. It then deletes those index records, and then the data record.

The method *executeModify* constructs the scan of records to be modified, as in the basic planner. Before modifying each record, the method first adjusts the index of the modified field, if it exists. In particular, it deletes the old index record and inserts a new one.

The methods to create tables, views, and indexes are the same as in the basic planner.

In order to get SimpleDB to use the index update planner, you must change the method *planner* in class *SimpleDB*, so that it creates an instance of *IndexUpdatePlanner* instead of *BasicUpdatePlanner*.

21.7 Chapter Summary

- Given a field A of table T, an *index* on A is a collection of records, one index record for each record of T. Each index record contains two fields: its *dataval*, which is the A-value of the corresponding record of T, and its *datarid*, which is the rid of the corresponding record.

- An index is able to improve the efficiency of *select* and *join* operations. Instead of scanning each block of the data table, the system can do the following:
    - *search* the index to find all index records having the selected dataval.

  - for each index record found, use its datarid to access the desired data record.

  In this way the database system is able to access only the data blocks that contain matching records.

- Indexes are implemented so that searches require very few disk accesses. We discussed three index implementation strategies: *static hashing, extendable hashing*, and *B-trees*.

- Static hashing stores index records in a fixed number of *buckets*, where each bucket corresponds to a file. A *hash function* determines the bucket assigned to each index record. To find an index record using static hashing, the index manager hashes its search key and examines that bucket. If an index contains B blocks and N buckets, then each bucket is about B/N blocks long, and so traversing a bucket requires about B/N block accesses.

- Extendable hashing allows buckets to share blocks. This improves on static hashing, because it allows for very many buckets without an especially large index file. Block sharing is achieved by means of a *bucket directory*. The bucket directory can be thought of as an array *Dir* of integers; if an index record hashes to bucket b, then that record will be stored in block *Dir[b]* of the bucket file. When a new index record does not fit in its block, then the block *splits*, the bucket directory is updated, and the block's records are rehashed.

- A B-tree stores its index records in a file sorted on their dataval. A B-tree also has a file of directory records. Each index block has a corresponding directory record, which contains the dataval of the first index record in the block and a reference to that block. These directory records level 0 of the B-tree directory. Similarly, each directory block has its own directory record, which is stored in the next level of the directory. The top level consists of a single block, which is called the *root* of the B-tree. Given a dataval, we can search the directory by examining one block at each level of the directory tree; this search leads us to the index block containing the desired index records.

- B-tree indexes are exceptionally efficient. Any desired data record can be retrieved in no more than 5 disk accesses, unless its table is unusually large. If a commercial database system implements only one indexing strategy, it almost certainly uses a B-tree.

21.8 Suggested Reading

In this chapter we treated indexes as implementations of the relational operators *select* and *join*. The article [Sieg and Sciore 1990] shows how an index can be treated as a special type of table, and how *indexselect* and *indexjoin* can be treated as relational algebra operators. This approach allows the planner to use indexes in a much more flexible way.

B-trees and hash files are general-purpose index structures, which work best when the query has a single selective search key. They do not work so well when queries have multiple search keys, such as in geographic and spatial databases. (For example, a B-tree cannot help with a query such as "find all restaurants within 2 miles of my home".) *Multidimensional indexes* have been developed to deal with such databases. The article [Gaede and Gunther 1998] provides a survey of these indexes.

The cost of a B-tree search is determined by the height of the B-tree, which is determined by the size of the index and directory records. The article [Bayer and Unteraurer 1977] gives techniques for reducing the size of these records. For example, if the datavals in a leaf node are strings and these strings have a common prefix, then we can store this prefix once at the beginning of the page, and store only the suffixes of the datavals with each index record. Moreover, there usually is no need to store the entire dataval in a directory record; we really only need to store the prefix of that dataval sufficient to determine which child to choose.

The article [Graefe 2004] describes a novel implementation of B-trees, in which nodes are never overridden; instead, updates to nodes cause new nodes to be created. The article demonstrates that this implementation results in faster updates at the cost of slightly slower reads.

This chapter has focused exclusively on how to minimize the number of disk accesses performed by a B-tree search. Although the CPU cost of a B-tree search is less important, it is often significant, and needs to be considered by commercial implementations. The article [Lomet 2001] discusses how to structure B-tree nodes in order to minimize search. The article [Chen et al 2002] shows how to structure B-tree nodes in order to maximize CPU cache performance.

In this chapter we also did not consider the issue of how to lock the nodes of a B-tree. SimpleDB simply locks a B-tree node the same as any other data block, and holds the lock until the transaction completes. However, it turns out that B-trees do not need to satisfy the lock protocol of Chapter 14 in order to guarantee serializability; instead, locks can be released early. The article [Bayer and Schkolnick 1977] addresses this issue.

Web search engines keep databases of web pages, which are primarily text. Queries on these databases tend to be based on string and pattern matching, for which traditional indexing structures are basically useless. Text-based indexing methods are treated in [Faloutsos 1985].

An unusual indexing strategy stores a bitmap for each field value; the bitmap contains one bit for each data record, and indicates whether the record contains that value. One interesting thing about bitmap indexes is that they can be easily intersected to handle multiple search keys. The article [O'Neil and Quass 1997] explains how bitmap indexes work.

Chapter 15 assumed that tables are stored sequentially, and are basically unorganized. However, it is also possible to organize a table according to a B-tree, hash file, or any other indexing strategy. There are some complications: for example, a B-tree record may move to another block when its block splits, which means that record ids must be handled carefully; furthermore, the indexing strategy must also support sequential scans through the table (and in fact, the entire *Scan* and *UpdateScan* interfaces). But the basic principles hold. The article [Batory 1982] describes how complex file organizations can be constructed out of the basic indexing strategies.

21.9 Exercises

CONCEPTUAL EXERCISES

21.1 Does it ever make sense to create an index for another index? Explain.

21.2 Assume that blocks are 120 bytes and that the DEPT table has 60 records. For each field of DEPT, calculate how many blocks are required to hold the index records.

21.3 The interface *Index* contains a method *delete*, which deletes the index record having a specified dataval and datarid. Would it be useful to also have a method *deleteAll*, which deletes all index records having a specified dataval? How and when would the planner use such a method?

21.4 Consider a query that joins two tables, such as

```
select SName, DName
from STUDENT, DEPT
where MajorId = DId
```

Suppose STUDENT contains an index on *MajorId*, and DEPT contains an index on *DId*. There are two ways to implement this query using an index join, one way for each index. Using the cost information from Figure 16-7, compare the cost of these two plans. What general rule can you conclude from your calculation?

21.5 The example of extendable hashing in Section 21.4 stopped after the insertion of only seven records. Continue the example, inserting records for employees having id 28, 9, 16, 24, 36, 48, 64, and 56.

21.6 In an extendable hashed index, consider an index block having local depth *l*. Show that the number of every bucket that points to this block has the same rightmost *l* bits.

21.7 In extendable hashing the bucket file increases when blocks split. Develop an algorithm for deletion that allows two split blocks to be coalesced. How practical is it?

21.8 Consider an extendable hash index such that 100 index records fit in a block. Suppose that the index is currently empty.
a) How many records can be inserted before the global depth of the index becomes 1?

b) How many records can be inserted before the global depth becomes 2?

21.9 Suppose that an insertion into an extendable hash index just caused its global depth to increase from 3 to 4.
a) How many entries will the bucket directory have?
b) How many blocks in the bucket file have exactly one directory entry pointing to them?

21.10 In this chapter we analyzed the number of block accesses required to search an index on *SName*, both for static hashing (in Section 21.2.1) and for B-trees (in Section 21.4.3). This exercise asks you to do a similar analysis for extendable hashing. In particular, assume that the block size is 4K bytes, which means that 1024 bucket directory values fit in a block; also assume that 157 index records fit in a block.
a) What is the maximum number of records that the index can have and still be searched in 2 block accesses?
b) What is the maximum number of records that the index can have and still be searched in 3 block accesses?

21.11 Suppose we create a B-tree index for *SId*. Assuming that 3 index records and 3 directory records fit into a block, draw a picture of the B-tree that results from inserting records for students 8, 12, 1, 20, 5, 7, 2, 28, 9, 16, 24, 36, 48, 64, and 56.

21.12 Consider the statistics of Figure 16-7, and suppose that we have a B-tree index on field *StudentId* of ENROLL. Assume that 100 index or directory records fit in a block.
a) How many blocks are in the index file?
b) How many blocks are in the directory file?

21.13 Consider again a B-tree index on field *StudentId* of ENROLL, and assume that 100 index or directory records fit in a block. Suppose that the index is currently empty.
a) How many insertions will cause the root to split (into a level-1 node)?
a) How many insertions will cause the root to split again (into a level-2 node)?

21.14 Consider the SimpleDB implementation of B-trees.
a) What is the maximum number of buffers that will be pinned simultaneously during an index scan?
b) What is the maximum number of buffers that will be pinned simultaneously during an insertion?

21.15 The SimpleDB indexscan implementation requires that the selection predicate be an equality comparison, as in "*GradYear = 1999*". In general, however, an index could be used with a range predicate, such as "*GradYear > 1999*".
a) Explain conceptually how a B-tree index on *GradYear* could be used to implement the following query:

```
select SName from STUDENT where GradYear > 1999
```

b) What revisions need to be made to the SimpleDB B-tree code in order to support your answer to part (a)?
c) Suppose that the database contains a B-tree index on *GradYear*. Explain why this index might not be useful for implementing the query. When would it be useful?
d) Explain why static and extendable hash indexes are never useful for this query.

PROGRAMMING EXERCISES

21.16 The methods *executeDelete* and *executeUpdate* of the SimpleDB update planner use a select scan to find the affected records. Another possibility is to use an index select scan, if an appropriate index exists.
a) Explain how the planner algorithms would have to change.
b) Implement these changes in SimpleDB.

21.17 Implement extendable hashing. Choose a maximum depth that creates a directory of at most 2 disk blocks.

21.18 Consider the following modification to index records: Instead of containing the rid of the corresponding data record, the index record only contains the block number where the data record lives. Thus there may be fewer index records than data records – If a data block contains multiple records for the same search key, a single index record would correspond to all of them.
a) Rewrite the code for *LookupScan* to accommodate this modification.
b) Explain why this modification could result in fewer disk accesses during index-based queries.
c) How do the index delete and insert methods have to change in order to accommodate this modification? Do they require more disk accesses than the existing methods? Write the necessary code, for both B-trees and static hashing.
d) Do you think that this modification is a good idea?

21.19 Many commercial database systems allow indexes to be specified from within the SQL *create table* statement. For example, the syntax of MySQL looks like this:

```
    create table T (A int, B varchar(9), index(A), C int, index(B))
```

That is, items of the form *index(<field>)* can appear anywhere inside the list of field names.
a) Revise the SimpleDB parser handle this additional syntax.
b) Revise the SimpleDB planner to create the appropriate plan.

21.20 One of the problems with the update planner method *executeCreateIndex* is that the newly-created index is empty, even if the indexed table contains records. Revise the method so that it automatically inserts an index record for every existing record in the indexed table.

21.21 Revise SimpleDB so that it has a *drop index* statement. Create your own syntax, and modify the parser and planner appropriately.

21.22 Revise SimpleDB so that a user can specify the type of a newly-created index.
a) Develop a new syntax for the create index statement, and give its grammar.
b) Modify the parser (and possibly the lexer) to implement your new syntax.

21.23 Implement static hashing using a single index file. The first N blocks of this file will contain the first block of each bucket. The remaining blocks in each bucket will be chained together, using an integer stored in the block. (For example, if the value stored in block 1 is 173, then the next block in the chain is block 173. A value of -1 indicates the end of the chain.) For simplicity, you can devote the first record slot of each block to hold this chain pointer.

21.24 SimpleDB splits a B-tree block as soon as it becomes full. Another algorithm is to allow blocks to be full, and to split them during the *insert* method. In particular, as the code moves down the tree looking for the leaf block, it splits any full block it encounters.
a) Modify the code to implement this algorithm.
b) Explain how this code reduces the buffer needs of the *insert* method.

22
MATERIALIZATION AND SORTING
and the package simpledb.materialize

This chapter considers operator implementations that materialize their input records by saving them in temporary tables. This materialization allows an implementation to access records multiple times without recomputation, and to sort records efficiently. We shall examine how materialization can be used to obtain efficient implementations of the *sort*, *groupby* and *join* operators.

22.1 The Value of Materialization

Every operator implementation that we have seen so far has had the following characteristics:

- Records are computed one at a time, as needed, and are not saved.
- The only way to access previously-seen records is to recompute the entire operation from the beginning.

In this chapter we shall consider implementations that *materialize* their input. Such implementations compute their input records when they are first opened, and save the records in one or more temporary tables. We say that these implementations *preprocess* their input, because they look at their entire input before any output records have been requested. The purpose of this materialization is to improve the efficiency of the ensuing scan.

> *An operator implementation may choose to materialize its input in order to improve the efficiency of the resulting scan.*

For example, consider the *groupby* operator. An implementation of this operator needs to group records together, and to calculate aggregation functions for each group. The easiest way to determine the groups is to sort the input records on the grouping fields, because sorting causes the records in each group to be next to each other. A good implementation strategy is therefore to first materialize the input, saving the records in a temporary table sorted on the grouping fields. The calculation of the aggregation functions can then be performed by making a single pass through the temporary table.

Materialization is a two-edged sword. On one hand, using a temporary table can significantly improve the efficiency of a scan. On the other hand, creating the temporary table can be expensive:

- The implementation incurs block accesses when it writes to the temporary table.
- The implementation incurs block accesses when it reads from the temporary table.
- The implementation must preprocess its entire input, even if the JDBC client is interested in only a few of the records.

A materialized implementation is useful only when these costs are offset by the increased efficiency of the scan. In this chapter, we shall examine several operators which have highly efficient materialized implementations.

22.2 Temporary Tables

Materialized implementations store their input records in *temporary tables*. A temporary table differs from a regular table in three ways:

- A temporary table is not created using the table manager's *createTable* method and its metadata does not appear in the system catalog. In SimpleDB, each temporary table manages its own metadata and has its own *getTableInfo* method.
- Temporary tables are automatically deleted by the database system when they are no longer needed. In SimpleDB, the file manager deletes the tables during system initialization.
- The recovery manager does not log changes to temporary tables. There is no need to recover the previous state of a temporary table, because the table will never be used after its query has completed.

SimpleDB implements temporary tables via the class *TempTable*; see Figure 22-1.

```
public class TempTable {
   private static int nextTableNum = 0;
   private TableInfo ti;
   private Transaction tx;

   public TempTable(Schema sch, Transaction tx) {
      String tblname = nextTableName();
      ti = new TableInfo(tblname, sch);
      this.tx = tx;
   }

   public UpdateScan open() {
      return new TableScan(ti, tx);
   }

   public TableInfo getTableInfo() {
      return ti;
   }

   private static synchronized String nextTableName() {
      nextTableNum++;
      return "temp" + nextTableNum;
   }
}
```

Figure 22-1: The code for the SimpleDB class *TempTable*

The constructor creates an empty table and assigns it a unique name (of the form "*tempN*" for some integer N). The class contains two public methods. The method *open* opens a table scan for the table. And the method *getTableInfo* returns the temporary table's metadata.

22.3 Materialization

22.3.1 The *materialize* operator

We begin by introducing a new operator to the relational algebra, called *materialize*. The *materialize* operator has no visible functionality. It takes a query as its only argument, and its output records are exactly the same as its input records. That is:

```
materialize(Q) ≡ Q
```

The purpose of the *materialize* operator is to save the output of a subquery in a temporary table, so that those records do not need to be computed multiple times. For example, consider the query tree of Figure 22-2(a). Recall that the implementation of the *product* node examines every record of its right subtree for each record in the left subtree. Consequently, the implementation will access the records of the left subtree once, and the records of the right subtree many times.

[redacted]

(a) The original query

[redacted]

(b) Materializing the left and right sides of the product

Figure 22-2: Where should we use the *materialize* operator?

The problem with accessing the right-side records repeatedly is that they will need to be recalculated repeatedly. In particular, the implementation will need to read the entire ENROLL table multiple times, and each time it will search for records having a grade of 'A'. Using the statistics of Figure 16-7, we can calculate the cost of the product as follows: There are 900 students in the class of 2005. The pipelined implementation will read the entire 50,000-block ENROLL table for each of these 900 students, which is 45,000,000 block accesses of ENROLL. When we add the 4,500 STUDENT blocks that also need to get read, we have a total of 45,004,500 block accesses.

Now consider the query tree of Figure 22-2(b), which has two *materialize* nodes. Let's start by considering the node above the right-side selection. That node creates a temporary table containing all of the ENROLL records having a grade of 'A'. Each time that the *product* node requests a record from its right side, the *materialize* node will take the record directly from this temporary table, instead of searching ENROLL.

We can calculate the effect of this *materialize* node on the cost of the product as follows. The temporary table will be 14 times smaller than ENROLL, or 3,572 blocks. The *materialize* node needs 53,572 block accesses to create the table (50,000 accesses to read ENROLL, and 3,572 accesses to write the table). After the temporary table has been created, it will be read 900 times, for a total of 3,214,800 accesses. When we add the 4,500 STUDENT block accesses, to these costs, we get a combined total of 3,272,872 block accesses. This total is considerably better than the cost of the original query tree. (In fact, at 1 msec per block access, it reduces the query time by over 11 hours.) Note that the cost of creating the temporary table is miniscule compared to the savings it generates.

A *materialize* node is only useful when the node's output will be calculated repeatedly. For example, consider the left-side *materialize* node in Figure 22-2(b). That node will scan the STUDENT table, and create a temporary table containing all students in the class of 2005. The *product* node will then examine this temporary table once. However, the *product* node in the original query tree also examines the STUDENT table once. Since the STUDENT records are examined once in each case, the only effect of the left-side *materialize* node is that it also creates the temporary table. Consequently, the left-side *materialize* node actually increases the cost of the query tree.

22.3.2 The cost of materialization

Figure 22-3 depicts the structure of a query tree that contains a *materialize* node.

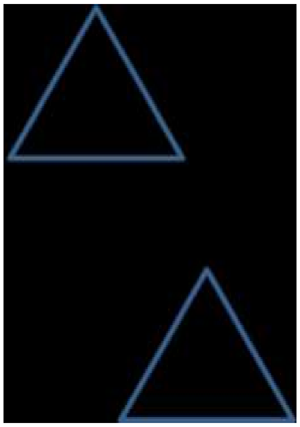


Figure 22-3: A query tree containing a *materialize* node

The input to the *materialize* node is the subquery denoted by T2. When a user opens a scan on the query T1, the root node will open its child nodes, and so on down the tree. When the *materialize* node is opened, it will preprocess its input. In particular, the node will open a scan for T2, evaluate it, save the output in a temporary table, and close the scan for T2. During the scan of query T1, the *materialize* node will respond to a request by accessing the corresponding record from its temporary table. Note that the subquery T2 is accessed once, to populate the temporary table; after that, it is no longer needed.

The cost associated with the materialize node can be divided into two parts: the cost of preprocessing the input, and the cost of executing the scan.

- The preprocessing cost is the cost of T2, plus the cost of writing the records to the temporary table.
- The scanning cost is the cost of reading the records from the temporary table.

Assuming that the temporary table is *B* blocks long, then these costs can be expressed as follows:

> *The cost of materializing a B-block table is:*
> - *Preprocessing cost = B + the cost of its input*
> - *Scanning cost = B*

22.3.3 Implementing the *materialize* operator

The *materialize* operator is implemented in SimpleDB by the class *MaterializePlan*, whose code appears in Figure 22-4.

```
public class MaterializePlan implements Plan {
   private Plan srcplan;
   private Transaction tx;

   public MaterializePlan(Plan srcplan, Transaction tx) {
      this.srcplan = srcplan;
      this.tx = tx;
   }

   public Scan open() {
      Schema sch = srcplan.schema();
      TempTable temp = new TempTable(sch, tx);
      Scan src = srcplan.open();
      UpdateScan dest = temp.open();
      // copy the input records to the temporary table
      while (src.next()) {
         dest.insert();
         for (String fldname : sch.fields())
            dest.setVal(fldname, src.getVal(fldname));
      }
      src.close();
      dest.beforeFirst();
      return dest;
   }

   public int blocksAccessed() {
      // create a dummy TableInfo object to calculate record length
      TableInfo ti = new TableInfo("dummy", srcplan.schema());
      double rpb = (double) (BLOCK_SIZE / ti.recordLength());
      return (int) Math.ceil(srcplan.recordsOutput() / rpb);
   }

   public int recordsOutput() {
      return srcplan.recordsOutput();
   }

   public int distinctValues(String fldname) {
      return srcplan.distinctValues(fldname);
   }

   public Schema schema() {
      return srcplan.schema();
   }
}
```

Figure 22-4: The code for the SimpleDB class *MaterializePlan*

The *open* method preprocesses its input, creating a new temporary table and copying the underlying records into it. The values for methods *recordsOutput* and *distinctValues* are the same as in the underlying plan. The method *blocksAccessed* returns the estimated size of the materialized table. This size is computed by calculating the records per block (RPB) of the new records and dividing the number of output records by this RPB.

Note that *blocksAccessed* does not include the preprocessing cost. The reason is that the temporary table is built once, but may be scanned multiple times. If we want to include the cost of building the table in our cost formulas, we need to add a new method (say, *preprocessingCost*) to the *Plan* interface, and to rework all of the various plan estimation formulas to include it. This task is left to Exercise 22.9. Alternatively, we can assume that the preprocessing cost is sufficiently insignificant, and ignore it in our estimates.

Also note that there is no *MaterializeScan* class. Instead, the method *open* returns a tablescan for the temporary table. The *open* method also closes the scan for the input records, because they are no longer needed.

22.4 Sorting

Section 4.2.3 introduced the *sort* operator of the relational algebra. The planner uses the *sort* operator to implement the *order by* clause of an SQL query. Moreover, sorting is also used to implement other relational algebra operators, such as *groupby* and *join*. We therefore need to be able to sort records efficiently. In this section we shall consider this problem and its SimpleDB solution.

22.4.1 Why *sort* needs to materialize its input

It is possible to implement the *sort* operator without using materialization. For example, consider the *sort* node in the query tree of Figure 22-5. The input to this node is the set of students and their majors, and the output is sorted by student name. Let’s assume for simplicity that no two students have the same name, so that the input records have distinct sort values.

Figure 22-5: A query tree containing a *sort* node

In a non-materialized implementation of *sort*, the method *next* will position the scan at the input record having the next largest *SName*-value. To do so, the method will have to compute all of the input records twice: first to find the next largest value, and then to move to the record having that value. Although such an implementation is possible, it would be exceptionally inefficient and totally impractical for large tables.

In a materialized implementation of *sort*, the method *open* will preprocess the input records, saving them in sorted order in a temporary table. Each call to *next* will simply retrieve the next record from the temporary table. Note how this implementation produces a very efficient scan at the cost of some initial preprocessing. If we assume that creating and sorting a temporary table can be performed relatively efficiently (which it can), then this materialized implementation will be considerably less expensive than the non-materialized one.

22.4.2 The basic mergesort algorithm

The standard sorting algorithms taught in beginning programming courses (such as insertion sort and quicksort) are called *internal* sorting algorithms, because they require all of the records to be in memory at the same time. A database system, however, cannot assume that a table will fit completely into memory; thus it must use *external* sorting algorithms. The simplest and most common external sorting algorithm is called *mergesort*.

The mergesort algorithm is based on the concept of a *run*.

A run is a sorted portion of a table.

An unsorted table has several runs; a sorted table has exactly one run. For example, suppose we want to sort students by their ID, and the *SId*-values of the STUDENT table are currently in the following order:

```
2   6   20   4   1   16   19   3   18
```

This table contains four runs. The first run contains the IDs [2, 6, 20], the second contains [4], the third contains [1, 16, 19], and the fourth [3, 18].

Mergesort works in two phases. The first phase, called *split*, scans the input records and places each run into its own temporary table. The second phase, called *merge*, repeatedly merges these runs until a single run remains; this final run is the sorted table.

The merge phase works as a sequence of *iterations*. During each iteration, the current set of runs is divided into pairs; each pair of runs is then merged into a single run. These resulting runs then form the new current set of runs. This new set will contain half as many runs as the previous one. The iterations continue until the current set contains a single run.

As an example of mergesort, let’s sort the above STUDENT records. The split phase identifies the four runs and stores each one in a temporary table:

```
Run 1:  2   6   20
Run 2:  4
Run 3:  1   16   19
Run 4:  3   18
```

In the first iteration of the merge phase, we merge runs 1 and 2 to produce run 5, and then merge runs 3 and 4 to produce run 6:

```
Run 5:  2   4   6   20
Run 6:  1   3   16  18  19
```

In the second iteration, we merge runs 5 and 6 to produce run 7:

```
Run 7:  1   2   3   4   6   16   18   19   20
```

We now have just one run, so we stop. Note that we needed just two merge iterations to sort the table.

Suppose that a table has 2^N initial runs. Each merge iteration transforms pairs of runs into a single run; that is, it reduces the number of runs by a factor of 2. Thus it will take *N* iterations to sort the file: the first iteration will reduce it to 2^{N-1} runs, the second to 2^{N-2} runs, and the N^{th} to 2^0=1 run. In general, a table with *R* initial runs will be sorted in log_2R merge iterations.

22.4.3 Improving the mergesort algorithm

There are three ways to improve the efficiency of this basic mergesort algorithm:

- We can increase the number of runs that we merge at a time.
- We can reduce the number of initial runs.
- We can avoid writing the final, sorted table.

This section examines these improvements.

Increasing the number of runs in a merge

Instead of merging pairs of runs, the algorithm could merge 3 runs at a time, or even more. Suppose that the algorithm merges *k* runs at a time. Then it would open a scan on each of *k* temporary tables. At each step, it looks at the current record of each scan, copies the lowest-valued one to the output table, and moves to the next record of that scan. This step is repeated until all records have been copied to the output table.

Merging multiple runs at a time reduces the number of iterations needed to sort the table. If the table starts with *R* initial runs and we merge *k* runs at a time, then we will need only log_kR iterations to sort the file.

How do we know what value of *k* to use? Why not just merge all of the runs in a single iteration? The answer depends on how many buffers are available. In order to merge *k* runs, we need $k+1$ buffers: one buffer for each of the *k* input scans, and one buffer for the output scan. For now, we assume that the algorithm picks an arbitrary value for *k*. In Chapter 23, we shall see how to pick the best value for *k*.

Reducing the number of initial runs

If we want to reduce the number of initial runs, then we need to increase the number of records per run. There are two algorithms that we can use.

The first algorithm is shown in Figure 22-6. That algorithm ignores the runs generated by the input records, and instead creates runs that are always one block long. It works by repeatedly storing a block's worth of input records in a temporary table. Recall that this block of records will be located in a buffer page in memory. The algorithm can therefore use an in-memory sorting algorithm (such as quicksort) to sort these records, without incurring any disk accesses. After sorting the block of records into a single run, it saves that block to disk.

```
Repeat until there are no more input records:
    1. Read a block's worth of input records into a new temporary table.
    2. Sort those records, using an in-memory sorting algorithm.
    3. Save the one-block temporary table to disk.
```

Figure 22-6: An algorithm to create initial runs that are exactly one block long

The second algorithm is similar, but it uses an additional block of memory as a "staging area" for input records. The code for this algorithm appears in Figure 22-7.

```
1. Fill the one-block staging area with input records.
2. Start a new run.
3. Repeat until the staging area is empty:
   a. If none of the records in the staging area fit into the current run, then:
          Close the current run, and start a new one.
   b. Choose the record from the staging area having the lowest value
      that is higher than the last record in the current run.
   c. Copy that record to the current run.
   d. Delete that record from the staging area.
   e. Add the next input record (if there is one) to the staging area.
4. Close the current run.
```

Figure 22-7: An algorithm to create large initial runs

The algorithm begins by filling the staging area with records. For as long as possible, it repeatedly deletes a record from the staging area, writes it to the current run, and adds another input record to the staging area. This procedure will stop when all of the records in the staging area are smaller than the last record in the run. In this case, the run is closed, and a new run is begun.

The advantage to using a staging area is that we keep adding records to it, which means that we always get to choose the next record in the run from a block-sized applicant pool. Thus each run will most likely contain more than a block's worth of records.

Let's do an example that compares these two ways of creating initial runs. Consider again the previous example, in which we sorted STUDENT records by their *SId* values. Assume that a block can hold three records, and that the records are initially in the following order:

```
2   6   20   4   1   16   19   3   18
```

These records happen to form four runs, as illustrated earlier. Suppose that we use the algorithm in Figure 22-6 to reduce the number of initial runs. Then we would read the records in groups of three, sorting each group individually. We therefore wind up with three initial runs, as follows:

```
Run 1:  2   6   20
Run 2:  1   4   16
Run 3:  3  18   19
```

Suppose instead that we use the algorithm in Figure 22-7 to reduce the number of runs. We begin by reading the first three records into the staging area.

```
Staging area:  2   6   20
Run 1:
```

We next choose the smallest value, 2, add it to the run, remove it from the staging area, and read the next record into the staging area.

```
Staging area:  6   20   4
Run 1:  2
```

The next smallest value is 4, so we add that value to the run, remove it from the staging area, and read in the next input value.

```
Staging area: 6   20   1
Run 1:  2  4
```

Here, the smallest value is 1, but that value is too small to be part of the current run. Instead, the next viable value is 6, so we add it to the run, and read the next input value into the staging area.

```
Staging area: 20  1  16
Run 1:  2  4  6
```

Continuing, we shall add 16, 19 and 20 to the run. At this point, the staging area consists entirely of records that cannot be added to the run.

```
Staging area: 1  3  18
Run 1:  2  4  6  16  19  20
```

We therefore begin a new run. Since there are no more input records, this run will contain the three records in the staging area.

```
Staging area:
Run 1:  2   4   6  16  19  20
Run 2:  1   3  18
```

Note that this algorithm produces only two initial runs in this case, and that the first run is two blocks long.

Don’t write the final sorted table

Recall that every materialized implementation has two stages: a preprocessing stage, in which the input records are materialized into one or more temporary tables; and a scanning stage, which uses the temporary tables to determine the next output record.

In the basic mergesort algorithm, the preprocessing stage creates a sorted temporary table, and the scan reads from that table. This is a simple strategy, but it is not optimal.

Instead of creating a sorted temporary table, suppose that the preprocessing stage stops before the final merge iteration; that is, it stops when the number of temporary tables is $\leq k$. The scanning stage would take these *k* tables as input, and perform the final merge itself. In particular, the stage would open a scan for each of these *k* tables. Each call to the method *next* would examine the current record of these scans, and choose the record having the smallest sort value.

Note that the scanning stage only needs to keep track of which of the *k* scans contains the current record at each point in time. We call this scan the *current scan*. When the client requests the next record, the implementation moves to the next record in the current scan, determines the scan containing the lowest record, and assigns that scan to be the new current scan.

To summarize, the job of the scanning stage is to return records in sorted order, as if they were stored in a single, sorted table. However, it does not need to actually create that table. Instead, it uses the *k* tables it receives from the preprocessing stage. Thus the block accesses needed to write (and read) the final, sorted table can be avoided.

22.4.4 The cost of mergesort

Let’s calculate the cost of sorting, using analysis similar to what we did for the *materialize* operator. Figure 22-8 depicts the structure of a query tree that contains a *sort* node.

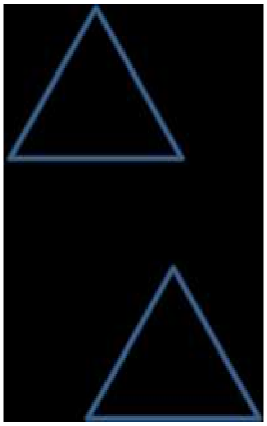


Figure 22-8: A query tree containing a *sort* node

The cost associated with the *sort* node can be divided into two parts: the preprocessing cost and the scanning cost.

- The preprocessing cost is the cost of T2, plus the cost of splitting the records, plus the cost of all but the last merge iteration.
- The scanning cost is the cost of performing the final merge from the records of the temporary tables.

In order to be more specific, we shall make the following assumptions:

- We merge *k* runs at a time.
- There are *R* initial runs.
- The materialized input records require *B* blocks.

The split phase writes to each of the blocks once, so splitting requires *B* block accesses, plus the cost of the input. The records can be sorted in log_kR iterations. One of those iterations will be performed by the scanning stage, and the rest by the preprocessing stage. During each preprocessing iteration, the records of each run will be read once and written once; thus the iteration requires *2B* block accesses. During the scanning stage, the records in each run will be read once, for a cost of *B* block accesses. Putting these values together and simplifying, we get the following cost formulas:

The cost of sorting a B-block table, having R initial runs, by merging k runs at a time, is:

- *Preprocessing cost =* $2Blog_kR$ *- B + the cost of its input*
- *Scanning cost = B*

For a concrete example, suppose we want to sort a 1,000-block stored table having 1-block long initial runs (that is, *B*=*R*=1,000). If we merge 2 runs at a time, then we need 10 merge iterations to completely sort the records (because $\log_2 1000 \leq 10$). The above formula states that it takes 20,000 block accesses to preprocess the records, plus another

1,000 for the final scan. If we merge 10 runs at a time (that is, *k*=10), then a we would need only 3 iterations, and preprocessing would require only 6,000 block accesses.

Continuing this example, suppose that we merge 1,000 runs at a time (that is, *k*=1,000). Then log_kR=1, so the preprocessing cost is *B* plus the cost of the input, or 2,000 block accesses. Note that the cost of sorting in this case is identical to the cost of materialization. The reason for this efficiency is that the preprocessing stage does not need to perform any merging, because the split phase already results in *k* runs. The cost of preprocessing is therefore the cost of reading the table and splitting the records, which is *2B* block accesses.

22.4.5 Implementing mergesort

The SimpleDB class *SortPlan* sorts records; its code appears in Figure 22-9.

```
public class SortPlan implements Plan {
   private Plan p;
   private Transaction tx;
   private Schema sch;
   private RecordComparator comp;

   public SortPlan(Plan p, Collection<String> sortfields,
                   Transaction tx) {
      this.p = p;
      this.tx = tx;
      sch = p.schema();
      comp = new RecordComparator(sortfields);
   }

   public Scan open() {
      Scan src = p.open();
      List<TempTable> runs = splitIntoRuns(src);
      src.close();
      while (runs.size() > 2)
         runs = mergeAllRuns(runs);
      return new SortScan(runs, comp);
   }

   public int blocksAccessed() {
      // does not include the one-time cost of sorting
      Plan mp = new MaterializePlan(p, tx); // for analysis only
      return mp.blocksAccessed();
   }

   public int recordsOutput() {
      return p.recordsOutput();
   }

   public int distinctValues(String fldname) {
      return p.distinctValues(fldname);
   }

   public Schema schema() {
      return sch;
   }

   private List<TempTable> splitIntoRuns(Scan src) {
      List<TempTable> temps = new ArrayList<TempTable>();
      src.beforeFirst();
      if (!src.next())
         return temps;
      TempTable currenttemp = new TempTable(sch, tx);
      temps.add(currenttemp);
      UpdateScan currentscan = currenttemp.open();
      while (copy(src, currentscan))
         if (comp.compare(src, currentscan) < 0) {
            // start a new run
            currentscan.close();
            currenttemp = new TempTable(sch, tx);
            temps.add(currenttemp);
            currentscan = (UpdateScan) currenttemp.open();
         }
      currentscan.close();
      return temps;
   }

   private List<TempTable> mergeAllRuns(List<TempTable> runs) {
      List<TempTable> result = new ArrayList<TempTable>();
      while (runs.size() > 1) {
         TempTable p1 = runs.remove(0);
         TempTable p2 = runs.remove(0);
         result.add(mergeTwoRuns(p1, p2));
      }
      if (runs.size() == 1)
         result.add(runs.get(0));
      return result;
   }

   private TempTable mergeTwoRuns(TempTable p1, TempTable p2) {
      Scan src1 = p1.open();
      Scan src2 = p2.open();
      TempTable result = new TempTable(sch, tx);
      UpdateScan dest = result.open();

      boolean hasmore1 = src1.next();
      boolean hasmore2 = src2.next();
      while (hasmore1 && hasmore2)
         if (comp.compare(src1, src2) < 0)
            hasmore1 = copy(src1, dest);
         else
            hasmore2 = copy(src2, dest);

      if (hasmore1)
         while (hasmore1)
            hasmore1 = copy(src1, dest);
      else
         while (hasmore2)
            hasmore2 = copy(src2, dest);
      src1.close();
      src2.close();
      dest.close();
      return result;
   }

   private boolean copy(Scan src, UpdateScan dest) {
      dest.insert();
      for (String fldname : sch.fields())
         dest.setVal(fldname, src.getVal(fldname));
      return src.next();
   }
```

Figure 22-9: The code for the SimpleDB class *SortPlan*

Let’s look first at the cost estimation methods. The methods *recordsOutput* and *distinctValues* are straightforward, because the sorted table contains the same records and value distribution as the underlying table. The method *blocksAccessed* estimates the number of block accesses required to iterate through the sorted scan, which is equal to the number of blocks in the sorted table. Since sorted and materialized tables are exactly the same size, this computation will be exactly the same as in *MaterializePlan*. Thus the method creates a "dummy" materialized plan for the sole purpose of calling its *blocksAccessed* method. Note that the preprocessing cost is not included in the *blocksAccessed* method, for the same reasons as with *MaterializePlan*.

The method *open* performs the basic mergesort algorithm. In particular, it merges two runs at a time (i.e. *k*=2), and does not try to reduce the number of initial runs. (Instead, Exercises 22.10 to 22.13 ask you to make these improvements.)

The private method *splitIntoRuns* performs the split phase of the mergesort algorithm, and method *doAMergeIteration* performs one iteration of the merge phase; this method is called repeatedly until it returns no more than two runs. At that point, the *open* method passes the list of runs to the *SortScan* constructor, which will handle the final merge iteration.

Method *splitIntoRuns* starts by creating a temporary table and opening a scan on it (the "destination scan"). The method then iterates through the input scan. Each input record is inserted into the destination scan. Each time a new run begins, the destination scan is closed, and another temporary table is created and opened. At the end of this method, several temporary tables will have been created, each containing a single run.

The method *doAMergeIteration* is given a list of the current temporary tables. It repeatedly calls the method *mergeTwoRuns* for each pair of temporary tables in the list, and returns a list containing the resulting (merged) temporary tables.

Method *mergeTwoRuns* opens a scan for each of its two tables, and creates a temporary table to hold the result. The method repeatedly chooses the smallest-valued record from the input scans, copying that record to the result. When one of the scans completes, then the remaining records of the other scan are added to the result.

The job of comparing records is performed by the class *RecordComparator*, whose code appears in Figure 22-10. The class compares the current records of two scans. Its *compare* method iterates through the sort fields, using the *compareTo* method to compare the values in each scan's current record. If all values are equal, then *compare* returns 0.

```
public class RecordComparator implements Comparator<Scan> {
   private Collection<String> fields;

   public RecordComparator(Collection<String> fields) {
      this.fields = fields;
   }

   public int compare(Scan s1, Scan s2) {
      for (String fldname : fields) {
         Constant val1 = s1.getVal(fldname);
         Constant val2 = s2.getVal(fldname);
         int result = val1.compareTo(val2);
         if (result != 0)
            return result;
      }
      return 0;
   }
}
```

Figure 22-10: The code for the SimpleDB class *RecordComparator*

The class *SortScan* implements the scan; its code appears in Figure 22-11.

```
public class SortScan implements Scan {
   private UpdateScan s1, s2=null, currentscan=null;
   private RecordComparator comp;
   private boolean hasmore1, hasmore2=false;
   private List<RID> savedposition;

   public SortScan(List<TempTable> runs, RecordComparator comp) {
      this.comp = comp;
      s1 = (UpdateScan) runs.get(0).open();
      hasmore1 = s1.next();
      if (runs.size() > 1) {
         s2 = (UpdateScan) runs.get(1).open();
         hasmore2 = s2.next();
      }
   }

   public void beforeFirst() {
      s1.beforeFirst();
      hasmore1 = s1.next();
      if (s2 != null) {
         s2.beforeFirst();
         hasmore2 = s2.next();
      }
   }

   public boolean next() {
      if (currentscan == s1)
         hasmore1 = s1.next();
      else if (currentscan == s2)
         hasmore2 = s2.next();

      if (!hasmore1 && !hasmore2)
```

```
         return false;
      else if (hasmore1 && hasmore2) {
         if (comp.compare(s1, s2) < 0)
            currentscan = s1;
         else
            currentscan = s2;
      }
      else if (hasmore1)
         currentscan = s1;
      else if (hasmore2)
         currentscan = s2;
      return true;
   }

   public void close() {
      s1.close();
      if (s2 != null)
         s2.close();
   }

   public Constant getVal(String fldname) {
      return currentscan.getVal(fldname);
   }

   public int getInt(String fldname) {
      return currentscan.getInt(fldname);
   }

   public String getString(String fldname) {
      return currentscan.getString(fldname);
   }

   public boolean hasField(String fldname) {
      return currentscan.hasField(fldname);
   }

   public void savePosition() {
      RID rid1 = s1.getRid();
      RID rid2 = s2.getRid();
      savedposition = Arrays.asList(rid1,rid2);
   }

   public void restorePosition() {
      RID rid1 = savedposition.get(0);
      RID rid2 = savedposition.get(1);
      s1.moveToRid(rid1);
      s2.moveToRid(rid2);
   }
}
```

Figure 22-11: The code for the SimpleDB class *SortScan*

The constructor expects a list containing one or two runs. It initializes the runs by opening their tables and moving to their first record. (If there is only one run, then the variable *hasmore2* is set to false, and the second run will never get considered.)

The variable *currentscan* points to the scan containing the most recent record in the merge. The *get* methods obtain their values from that scan. The method *next* moves to the next record of the current scan, and then chooses the lowest-value record from the two scans. The variable *currentscan* then points to that scan.

The class also has the two public methods *savePosition* and *restorePosition*. These methods allow a client (in particular, the mergejoin scan of Section 22.6) to move back to a previously-seen record, and continue the scan from there.

22.5 Grouping and Aggregation

22.5.1 The *groupby* algorithm

The *groupby* operator was introduced in Section 4.2.6. It takes three arguments: an input table, a collection of grouping fields, and a collection of aggregation functions. Its output table has one record for each group, and its schema has one field for each grouping field and for each aggregation function. The main issue in implementing *groupby* is how to determine the records in each group.

A non-materialized implementation of *groupby* would have a hard time determining the records in each group. For each input record, the implementation would have to examine every other input record to determine which ones belonged to the same group. Such an implementation would be woefully inefficient. A materialized implementation can do much better, by creating a temporary table in which the records are sorted on the grouping fields. The records in each group will then be next to each other, and so the implementation can calculate the information on every group by making a single pass through the sorted table. Figure 22-12 gives the algorithm.

```
1. Create a temporary table containing the input records, sorted by
   the grouping fields.
2. Move to the first record in the table.
3. Repeat until the temporary table is exhausted:
   a. Let the “group value” be the values of the grouping fields
      for the current record.
   b. For each record whose grouping field values equals the group value:
            Read the record into the group list.
   c. Calculate the specified aggregation functions for the records
      in the group list.
```

.

Figure 22-12: An algorithm to perform aggregation

22.5.2 The cost of *groupby*

The cost of the aggregation algorithm can be split into its preprocessing cost and its scanning cost. These costs are straightforward. The preprocessing cost is a sort, and the scanning cost is a single iteration through the sorted records. In other words, the cost of *groupby* is identical to that of *sort*.

22.5.3 Implementing *groupby*

SimpleDB implements uses the classes *GroupByPlan* and *GroupByScan* to implement the groupby algorithm; see Figures 22-13 and 22-14.

```
public class GroupByPlan implements Plan {
   private Plan p;
   private Collection<String> groupfields;
   private Collection<AggregationFn> aggfns;
   private Schema sch = new Schema();
   private Transaction tx;

   public GroupByPlan(Plan p, Collection <String> groupfields,
                    Collection<AggregationFn> aggfns, Transaction tx) {
      this.p = p;
      this.groupfields = groupfields;
      this.aggfns = aggfns;
      this.tx = tx;
      for (String fldname : groupfields)
         sch.add(fldname, p.schema());
      for (AggregationFn fn : aggfns)
         sch.addIntField(fn.fieldName());
   }

   public Scan open() {
      Plan sortplan = new SortPlan(p, groupfields, tx);
      Scan s = sortplan.open();
      return new GroupByScan(s, groupfields, aggfns);
   }

   public int blocksAccessed() {
      return p.blocksAccessed();
   }

   public int recordsOutput() {
      int numgroups = 1;
      for (String fldname : groupfields)
         numgroups *= p.distinctValues(fldname);
      return numgroups;
   }

   public int distinctValues(String fldname) {
      if (p.schema().hasField(fldname))
         return p.distinctValues(fldname);
      else
         return recordsOutput();
   }

   public Schema schema() {
      return sch;
   }
}
```

Figure 22-13: The code for the SimpleDB class *GroupByPlan*

```
public class GroupByScan implements Scan {
   private Scan s;
   private Collection<String> groupfields;
   private Collection<AggregationFn> aggfns;
   private GroupValue groupval;
   private boolean moregroups;

   public GroupByScan(Scan s, Collection<String> groupfields,
                      Collection<AggregationFn> aggfns) {
      this.s = s;
      this.groupfields = groupfields;
      this.aggfns = aggfns;
      beforeFirst();
   }

   public void beforeFirst() {
      s.beforeFirst();
      moregroups = s.next();
   }

   public boolean next() {
      if (!moregroups)
         return false;
      for (AggregationFn fn : aggfns)
         fn.processFirst(s);
      groupval = new GroupValue(s, groupfields);
      while((moregroups = s.next()) && groupval.sameGroupAs(s))
         for (AggregationFn fn : aggfns)
            fn.processNext(s);
      return true;
   }

   public void close() {
      s.close();
   }

   public Constant getVal(String fldname) {
      if (groupfields.contains(fldname))
         return groupval.getVal(fldname);
      for (AggregationFn fn : aggfns)
         if (fn.fieldName().equals(fldname))
            return fn.value();
      throw new RuntimeException("field " + fldname + " not found.");
   }

   public int getInt(String fldname) {
      return (Integer) getVal(fldname).asJavaVal();
   }

   public String getString(String fldname) {
      return (String) getVal(fldname).asJavaVal();
   }

   public boolean hasField(String fldname) {
      if (groupfields.contains(fldname))
         return true;
      for (AggregationFn fn : aggfns)
```

```
         if (fn.fieldName().equals(fldname))
            return true;
      return false;
   }
}
```

Figure 22-14: The code for the SimpleDB class *GroupByScan*

Method *open* in *GroupByPlan* creates and opens a sort plan for the input records. The resulting sort scan is passed into the constructor of *GroupByScan*. The groupby scan reads the records of the sort scan as needed. In particular, the method *next* reads the records in the next group each time it is called. This method will recognize the end of a group after it reads a record from another group (or when it detects that there are no more records in the sorted scan); consequently, each time *next* is called, the current record in the underlying scan will always be the first record in the next group.

The class *GroupValue* holds information about the current group; see Figure 22-15. A scan is passed into its constructor, together with the grouping fields. The field values of the current record define the group. The method *getVal* returns the value of a specified field. The *equals* method returns *true* when the two *GroupValue* objects have the same values for the grouping fields, and the *hashCode* method assigns a hash value to each *GroupValue* object.

```
public class GroupValue {
   private Map<String,Constant> vals;

   public GroupValue(Scan s, Collection<String> fields) {
      vals = new HashMap<String,Constant>();
      for (String fldname : fields)
         vals.put(fldname, s.getVal(fldname));
   }

   public Constant getVal(String fldname) {
      return vals.get(fldname);
   }

   public boolean equals(Object obj) {
      GroupValue gv = (GroupValue) obj;
      for (String fldname : vals.keySet()) {
         Constant v1 = vals.get(fldname);
         Constant v2 = gv.getVal(fldname);
         if (!v1.equals(v2))
            return false;
      }
      return true;
   }

   public int hashCode() {
      int hashval = 0;
      for (Constant c : vals.values())
         hashval += c.hashCode();
      return hashval;
   }
}
```

Figure 22-15: The code for the SimpleDB class *GroupValue*

SimpleDB implements each aggregation function (such as MIN, COUNT, etc.) as a class. An object of the class is responsible for keeping track of the relevant information about the records in a group, for calculating the aggregate value for this group, and for determining the name of the calculated field. These methods belong to the interface *AggregationFn*; see Figure 22-16. Method *processFirst* starts a new group using the current record as the first record of that group. Method *processNext* adds another record to the existing group.

```
public interface AggregationFn {
   void processFirst(Scan s);
   void processNext(Scan s);
   String fieldName();
   Constant value();
}
```

Figure 22-16: The code for the SimpleDB *AggregationFn* interface

An example of an aggregation function class is *MaxFn*, which implements MAX; see Figure 22-17. The client passes the name of the aggregated field into the constructor.

The object uses this field name to examine the field value from each record in the group, and it saves the maximum one in its variable *val*.

```
public class MaxFn implements AggregationFn {
   private String fldname;
   private Constant val;

   public MaxFn(String fldname) {
      this.fldname = fldname;
   }

   public void processFirst(Scan s) {
      val = s.getVal(fldname);
   }

   public void processNext(Scan s) {
      Constant newval = s.getVal(fldname);
      if (newval.compareTo(val) > 0)
         val = newval;
   }

   public String fieldName() {
      return "maxof" + fldname;
   }

   public Constant value() {
      return val;
   }
}
```

Figure 22-17: The code for the SimpleDB class *MaxFn*

22.6 Merge Joins

The SimpleDB basic query planner of Chapter 19 computes a join as the selection of a product. That is, it first computes the product of the two input tables, and then performs a selection using the join predicate. This strategy is very expensive, because the right-side table will be scanned once for every record in the left-side table[†]. In Chapter 21 we developed an efficient implementation for *join* in the special case when an index is available. In this section we examine an efficient materialized implementation for *join* in the case when its join predicate is an equijoin.

22.6.1 The mergejoin algorithm

The *join* operator was introduced in Section 4.2.7. It takes three arguments: two input tables, and an arbitrary join predicate. The most common join predicate is an *equijoin*; that is, it is of the form "A=B" where A and B are join fields. In this case the join has a materialized implemention called *mergejoin*. The mergejoin algorithm appears in Figure 22-18.

[†] Actually, it is more accurate to say that our naïve implementation of *product* is expensive. Chapter 23 develops a very efficient implementation of *product*, which can sometimes be the best way to join two tables.

```
1. For each input table:
        Sort the table, using its join field as the sort field.
2. Scan the sorted tables in parallel, looking for matches.
```

Figure 22-18: The *mergejoin* algorithm

Let's look at step 2 of the algorithm in more detail. Assume for the moment that the table on the left side of the join has no duplicate values in its join field. The procedure is actually similar to a product scan. We scan the left-side table once. For each left-side record, we search the right-side table looking for matching records. However, because the records are sorted, we know two things:

- The matching right-side records must begin after the records for the previous left-side record.
- The matching records are next to each other in the table.

Consequently, each time a new left-side record is considered, it suffices to continue scanning the right-side table from where we left off, and to stop when we reach a join value greater than the left-side join value. That is, the right-side table need only be scanned once.

For example, consider the following query:

```
join(DEPT, STUDENT, DId=MajorId)
```

The first step of the mergejoin algorithm creates temporary tables to hold the contents of DEPT and STUDENT, sorted on fields *DId* and *MajorId* respectively. Figure 22-19 shows the resulting sorted tables, using the sample records from Figure 1-1, but extended with a new department (the Basketry department, *DId*=18).

DEPT	**DId**	**DName**
	10	compsci
	18	basketry
	20	math
	30	drama

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

Figure 22-19: The sorted DEPT and STUDENT tables

The second step of the algorithm is to scan through the sorted tables. The current DEPT record is department 10. We scan STUDENT, finding a match at the first three records. When we move to the fourth record (for Amy), we discover a different *MajorId*-value, and so we know we are done with department 10. We move to the next DEPT record (for the Basketry department), and compare its *DId*-value with the *MajorId*-value of the current STUDENT record (i.e., Amy). Since Amy's *MajorId*-value is larger, we know that there are no matches for that department, and therefore move to the next DEPT record (for the Math department). This record matches Amy's record, as well as the next three STUDENT records. As we move through STUDENT, we eventually get to Bob's record, which does not match with the current department. So we move to the next DEPT record (for the Drama department), and continue our search through STUDENT, where the records for Bob and Art match. The join can finish as soon as one of the tables has run out of records.

22.6.2 Dealing with duplicate left-side values

What happens if the left side of a mergejoin has duplicate join values? Recall that we move to the next left-side record when we read a right-side record that no longer matches. If the next left-side record has the same join value, then we need to move back to the first matching right-side record. That is, all of the right-side blocks containing matching records will have to be re-read, potentially increasing the cost of the join.

Fortunately, duplicate left-side values rarely occur. Most joins in a query tend to be relationship joins – that is, the join fields correspond to a key-foreign key relationship. For example, the above join of STUDENT and DEPT is a relationship join, since *DId* is a key of DEPT and *MajorId* is its foreign key. Since keys and foreign keys are declared when the table is created, the query planner can use this information to recognize a relationship join, and to thereby ensure that the table having the key is on the left side of the mergejoin.

22.6.3 The cost of mergejoin

The preprocessing phase of the *mergejoin* algorithm sorts each input table. The scanning phase iterates through the sorted tables. If there are no duplicate left-side values, then each sorted table gets scanned once. If there are duplicate left-side values, then the corresponding records in the right-side scan will be read multiple times.

Let's calculate the cost of the above example, which joins the STUDENT and DEPT tables. We shall use the statistics of Figure 16-7. Assume that we merge pairs of runs, and each initial run is 1-block long. The preprocessing cost includes sorting the 4,500-block STUDENT table (for 9,000 * $\log_2(4{,}500) - 4{,}500 = 112{,}500$ block accesses, plus 4,500 for the cost of the input), and sorting the 2-block DEPT table (for 4 * $\log_2(2) - 2 = 2$ block accesses, plus 2 for the cost of the input). The total preprocessing cost is thus 117,004 block accesses. The scanning cost is the sum of the sizes of the sorted tables, which is 4,502 block accesses. The total cost of the join is thus 121,506 block accesses.

Compare this cost with the cost of performing the join as a product followed by a selection, as in Chapter 17. That cost formula is $B_1 + R_1*B_2$, which comes to 184,500 block accesses.

22.6.4 Implementing mergejoin

The SimpleDB classes *MergeJoinPlan* and *MergeJoinScan* implement the mergejoin algorithm. The code for *MergeJoinPlan* appears in Figure 22-20.

```
public class MergeJoinPlan implements Plan {
   private Plan p1, p2;
   private String fldname1, fldname2;
   private Schema sch = new Schema();

   public MergeJoinPlan(Plan p1, Plan p2, String fldname1,
                        String fldname2, Transaction tx) {
      this.fldname1 = fldname1;
      List<String> sortlist1 = Arrays.asList(fldname1);
      this.p1 = new SortPlan(p1, sortlist1, tx);

      this.fldname2 = fldname2;
      List<String> sortlist2 = Arrays.asList(fldname2);
      this.p2 = new SortPlan(p2, sortlist2, tx);

      sch.addAll(p1.schema());
      sch.addAll(p2.schema());
   }

   public Scan open() {
      Scan s1 = p1.open();
      SortScan s2 = (SortScan) p2.open();
      return new MergeJoinScan(s1, s2, fldname1, fldname2);
   }

   public int blocksAccessed() {
      return p1.blocksAccessed() + p2.blocksAccessed();
   }

   public int recordsOutput() {
      int maxvals = Math.max(p1.distinctValues(fldname1),
                             p2.distinctValues(fldname2));
      return (p1.recordsOutput() * p2.recordsOutput()) / maxvals;
   }

   public int distinctValues(String fldname) {
      if (p1.schema().hasField(fldname))
         return p1.distinctValues(fldname);
      else
         return p2.distinctValues(fldname);
   }

   public Schema schema() {
      return sch;
   }
```

Figure 22-20: The code for the SimpleDB class *MergeJoinPlan*

The method *open* opens a sort scan for each of the two input tables, using the specified join fields. It then passes these scans to the *MergeJoinScan* constructor.

The method *blocksAccessed* assumes that each scan will be traversed once. The idea is that even if there are duplicate left-side values, the matching right-side records will either be in the same block or a recently-accessed one. Thus it is likely that very few (or possibly zero) additional block accesses will be needed.

The method *recordsOutput* calculates the number of records of the join. This value will be the number of records in the product, divided by the number of records filtered out by the join predicate. Those formulas were given in Section 17.5.

The code for the method *distinctValues* is straightforward. Since the join does not increase or decrease field values, the estimate is the same as in the appropriate underlying query.

The code for *MergeJoinScan* appears in Figure 22-21.

```
public class MergeJoinScan implements Scan {
   private Scan s1;
   private SortScan s2;
   private String fldname1, fldname2;
   private Constant joinval = null;

   public MergeJoinScan(Scan s1, SortScan s2, String fldname1,
                        String fldname2) {
      this.s1 = s1;
      this.s2 = s2;
      this.fldname1 = fldname1;
      this.fldname2 = fldname2;
      beforeFirst();
   }

   public void beforeFirst() {
      s1.beforeFirst();
      s2.beforeFirst();
   }

   public void close() {
      s1.close();
      s2.close();
   }

   public boolean next() {
      boolean hasmore2 = s2.next();
      if (hasmore2 && s2.getVal(fldname2).equals(joinval))
         return true;

      boolean hasmore1 = s1.next();
      if (hasmore1 && s1.getVal(fldname1).equals(joinval)) {
         s2.restorePosition();
         return true;
      }

      while (hasmore1 && hasmore2) {
         Constant v1 = s1.getVal(fldname1);
         Constant v2 = s2.getVal(fldname2);
         if (v1.compareTo(v2) < 0)
            hasmore1 = s1.next();
         else if (v1.compareTo(v2) > 0)
            hasmore2 = s2.next();
         else {
            s2.savePosition();
            joinval  = s2.getVal(fldname2);
            return true;
         }
      }
      return false;
   }

   public Constant getVal(String fldname) {
      if (s1.hasField(fldname))
         return s1.getVal(fldname);
      else
         return s2.getVal(fldname);
   }

   public int getInt(String fldname) {
      if (s1.hasField(fldname))
         return s1.getInt(fldname);
      else
         return s2.getInt(fldname);
   }

   public String getString(String fldname) {
      if (s1.hasField(fldname))
         return s1.getString(fldname);
      else
         return s2.getString(fldname);
   }

   public boolean hasField(String fldname) {
      return s1.hasField(fldname) || s2.hasField(fldname);
   }
}
```

Figure 22-21: The code for the SimpleDB class *MergeJoinScan*

Method *next* performs the difficult work of looking for matches. A mergejoin scan keeps track of the most recent join value (in variable *joinval*). When *next* is called, it reads the next right-side record. If this record has a join value equal to *joinval*, a match is found and the method returns. If not, then the method moves to the next left-side record. If this record's join value equals *joinval*, then we have a duplicate left-side value. The method repositions the right-side scan to the first record having that join value, and returns.

Otherwise, the method repeatedly reads from the scan having the lowest join value, until either a match is found or a scan runs out. If a match is found, the variable *joinval* is set, and the current right-side position is saved. If a scan runs out, the method returns *false*.

22.7 Chapter Summary

- A *materialized* implementation of an operator preprocesses its underlying records, storing them in one or more temporary tables. Its scan methods are thus more efficient, because they only need to examine the temporary tables.

- Materialized implementations compute their input once and can take advantage of sorting. However, they must compute their entire input table even if the user is interested in only a few of those records. Although it is possible to write materialized implementations for any relational operator, a materialized implementation will be useful only if its preprocessing cost is offset by the savings of the resulting scan.

- The *materialize* operator creates a temporary table containing all of its input records. It is useful whenever its input is executed repeatedly, such as when it is on the right side of a *product* node.

- A database system uses an *external* sorting algorithm to sort its records into a temporary table. The simplest and most common external sorting algorithm is called *mergesort*. The mergesort algorithm splits the input records into runs, and then repeatedly merges the runs until the records are sorted.

- Mergesort is more efficient when the number of initial runs is smaller. A straightforward approach is to create initial runs that are one block long, by reading the input records into a block and then using an internal sorting algorithm to sort them. Another approach is to read input records into a one-block-long *staging area*, and to construct runs by repeatedly selecting the lowest-valued record in the area.

- Mergesort is also more efficient when it merges more runs at a time. The more runs that are merged, the fewer iterations that are needed. A buffer is needed to manage each merged run, so the maximum number of runs is limited by the number of available buffers.

- Mergesort requires $2B\log_k(R)-B$ block accesses (plus the cost of the input) to preprocess its input, where B is the number of blocks required to hold the sorted table, R is the number of initial runs, and k is the number of runs that are merged at one time.

- The implementation of the *groupby* operator sorts the records on the grouping fields, so that the records in each group are next to each other. It then calculates the information on each group by making a single pass through the sorted records.

- The *mergejoin* algorithm implements the *join* operator. It begins by sorting each table on its join field. It then scans the two sorted tables in parallel. Each call to the *next* method increments the scan having the lowest value.

22.8 Suggested Reading

File sorting has been an important (even crucial) operation throughout the history of computing, predating database systems by many years. There is an enormous literature on the subject, and numerous variations on mergesort that we have not considered. A comprehensive overview of the various algorithms appears in [Knuth 1998].

The SimpleDB *SortPlan* code is a straightforward implementation of the mergesort algorithm. The article [Graefe 2006] describes several interesting and useful techniques for improving upon this implementation.

The article [Graefe 2003] explores the duality between sort algorithms and B-tree algorithms. It shows how to use a B-tree to usefully store the intermediate runs of a mergesort, and how merge iterations can be used to create a B-tree index for an existing table.

Materialized algorithms are discussed in [Graefe 1993], and are compared with non-materialized algorithms.

22.9 Exercises

CONCEPTUAL EXERCISES

22.1 Consider the query tree of Figure 22-2(b).
a) Supose that there was only one student in the class of 2005. Is the right-hand *materialize* node worthwhile?
b) Supose that there were only two students in the class of 2005. Is the right-hand *materialize* node worthwhile?
c) Suppose that the right and left subtrees of the *product* node were swapped. Calculate the savings of materializing the new right-hand *select* node.

22.2 Consider the basic mergesort algorithm of Section 22.4.2. That algorithm merges the runs iteratively. Using the example of that section, it merged runs 1 and 2 to produce run 5, and runs 3 and 4 to produce run 6; then it merged runs 5 and 6 to produce the final run. Suppose instead that the algorithm merged the runs sequentially. That is, it merges runs 1 and 2 to produce run 5, then merges runs 3 and 5 to produce run 6, and then merges runs 4 and 6 to produce the final run.
a) Explain why the final run produced by this "sequential merging" will always require the same number of merges as with iterative merging.
b) Explain why sequential merging requires more (and usually many more) block accesses than iterative merging.

22.3 Consider the run-generation algorithms of Figures 22-6 and 22-7.

a) Suppose that the input records are already sorted. Which algorithm will produce the fewest initial runs? Explain.
b) Suppose that the input records are sorted in reverse order. Explain why the algorithms will produce the same number of initial runs.

22.4 Consider our university database and the statistics of Figure 16-7.
a) For each table, estimate the cost of sorting it using 2, 10, or 100 auxiliary tables. Assume that each initial run is one block long.
b) For each pair of tables that can be meaningfully joined, estimate the cost of performing a mergejoin (again, using 2, 10, or 100 auxiliary tables).

22.5 The method *splitIntoRuns* in the class *SortPlan* returns a list of *TempTable* objects. If the database is very large, then this list might be very long.
a) Explain how this list might be a source of unexpected inefficiency.
b) Propose a solution that would be better.

PROGRAMMING EXERCISES

22.6 Section 22.4.1 described a non-materialized implementation of sorting.
a) Design and implement the classes *NMSortPlan* and *NMSortScan*, which provide sorted access to records without the creation of temporary tables.
b) How many block accesses are required to fully traverse such a scan?
c) Suppose that a JDBC client wants to find the record having the minimum value for a field; the client does so by executing a query that sorts the table on that field and then choosing the first record. Compare the number of block accesses required to do this, using the materialized and non-materialized implementations.

22.7 When the server restarts, temporary table names begin again from 0. The SimpleDB file manager constructor deletes all temporary files.
a) Explain what problem will occur in SimpleDB if temporary table files were allowed to remain after the system restarts.
b) Instead of deleting all temporary files when the system restarts, the system could delete the file for a temporary table as soon as the transaction that created it has completed. Revise the SimpleDB code do this.

22.8 What problem occurs when *SortPlan* is asked to sort an empty table? Revise the code to fix the problem.

22.9 Revise the SimpleDB *Plan* interface (and all of its implementing classes) to have a method *preprocessingCost*, which estimates the one-time cost of materializing a table. Modify the other estimation formulas appropriately.

22.10 Revise the code for *SortPlan* so that it constructs initial runs of one block long, using the algorithm of Figure 22-5.

22.11 Revise the code for *SortPlan* so that it constructs initial runs using a staging area, using the algorithm of Figure 22-6.

22.12 Revise the code for *SortPlan* so that it merges 3 runs at a time.

22.13 Revise the code for *SortPlan* so that it merges k runs at a time, where the integer k is supplied in the constructor.

22.14 Revise the SimpleDB *Plan* classes so that they keep track of whether their records are sorted, and if so on what fields. Then revise the code for *SortPlan* so that it sorts the records only if necessary.

22.15 An *order by* clause in an SQL query is optional. If it exists, it consists of the two keywords "order" and "by", followed by a comma-separated list of field names.
a) Revise the SQL grammar of Chapter 18 to include *order by* clauses.
b) Revise the SimpleDB lexical analyzer and query parser to implement your syntax changes.
c) Revise the SimpleDB query planner to generate an appropriate *sort* operation for queries containing an *order by* clause.

22.16 SimpleDB only implements the aggregation functions COUNT and MAX. Add classes that implement MIN, AVG, and SUM.

22.17 Section 4.3.7 describes the syntax of SQL aggregation statements.
a) Revise the SQL grammar of Chapter 18 to include this syntax.
b) Revise the SimpleDB lexical analyzer and query parser to implement your syntax changes.
c) Revise the SimpleDB query planner to generate an appropriate *groupby* operation for queries containing a *group b*y clause.

22.18 Define a relational operator *nodups*, whose output table consists of those records from its input table, but with duplicates removed.
a) Write code for *NoDupsPlan* and *NoDupsScan*, similar to how *GroupByPlan* and *GroupByScan* are written.
b) Section 4.2.6 explained how duplicate removal can be performed by a *groupby* operator with no aggregation functions. Write code for *GBNoDupsPlan*, which implements *nodups* operator by creating the appropriate *GroupByPlan* object.

22.19 The keyword "distinct" can optionally appear in the select clause of an SQL query. If it exists, the query processor should remove duplicates from the output table.
a) Revise the SQL grammar of Chapter 18 to include the *distinct* keyword.
b) Revise the SimpleDB lexical analyzer and query parser to implement your syntax changes.
c) Revise the basic query planner to generate an appropriate *nodups* operation for *select distinct* queries.

22.20 Another way to sort a table on a single field is to use a B-tree index. The *SortPlan* constructor would first create an index for the materialized table on the sortfield. It then would add an index record into the B-tree for each data record. The records can then be read in sorted order by traversing the leaf nodes of the B-tree from the beginning.
a) Implement this version of *SortPlan*. (You will need to modify the B-tree code so that all index blocks are chained.)
b) How many block accesses does it require? Is it more or less efficient than using mergesort?

22.21 SimpleDB implements temporary tables as record files. Although it is convenient to use an already-existing implementation, it is not necessarily most efficient. Temporary tables do not need the full generality of recordfiles: they are modified only by appending records to the end, and do not need rids.
a) Give an API for an efficient implementation of temporary tables.
b) Implement this API. Modify the existing SimpleDB code to call the methods of your API.

23
EFFECTIVE BUFFER UTILIZATION
and the package simpledb.multibuffer

Different operator implementations have different buffer needs. For example, our pipelined implementation of *select* uses a single buffer very efficiently, and has no need for additional buffers. On the other hand, our materialized implementation of *sort* needs a buffer for each run that it merges at a time, and thus can be more efficient when it uses additional buffers.

In this chapter we consider the various ways in which operator implementations can use additional buffers, and give efficient multibuffer algorithms for *sort*, *product*, and *join*.

23.1 Buffer Usage in Query Plans

All of our relational algebra implementations have so far been very frugal when it comes to buffer usage. For example, each table scan pins one block at a time: when it finishes with the records in a block, it unpins that block before pinning the next one. The scans for the basic SimpleDB operators (*select*, *project*, *product*) do not pin any additional blocks. Consequently, given an *N*-table query, the SimpleDB basic query planner will produce a scan that requires *N* simultaneous pinned buffers.

The index implementations of Chapter 21 require only one buffer for each index. A static hash index implements each bucket as a file and scans it sequentially, pinning one block at a time. And a B-tree index works by pinning one directory block at a time, starting at the root. It scans the block to determine the appropriate child, unpins the block, and pins the child block, continuing until the leaf block is found.[†]

Now consider the materialized implementations of Chapter 22. The implementation of *materialize* requires one buffer for the temporary table, in addition to the buffers needed by the input query. The split phase of the *sort* implementation requires one or two additional buffers for the split phase (depending on whether we are using a staging area); and the merge phase requires *k+1* buffers: one buffer for each of the *k* runs being merged, and one buffer for the result table. And the implementations of *groupby* and *mergejoin* require no additional buffers beyond those used for sorting.

[†] This analysis is at least true for queries, which is what we are interested in here. Inserting a record into a B-tree may require several buffers to be pinned simultaneously, in order to handle block splitting and the recursive insertion of entries up the tree. Exercise 21.16 asked you to analyze the buffer requirements for insertions.

This analysis shows that, with the exception of sorting, the number of simultaneous buffers used by a query plan is roughly equal to the number of tables mentioned in the query; this number is usually less than 10 and almost certainly less than 100. The total number of available buffers is typically much larger. Server machines these days typically have at least 2GB of physical memory. If only a paltry 40MB of that is used for buffers, then the server would have 10,000 4K-byte buffers. A high-powered dedicated database server would more likely have 100,000 or more buffers. So even if the database system supports hundreds (or thousands) of simultaneous connections, there are still plenty of buffers available, if only the query plan were able to use them effectively. In this chapter we consider several operators – *sort*, *join*, and *product* – and how they can take advantage of this abundance of buffers.

23.2 MultiBuffer Sorting

Recall that the mergesort algorithm has two phases: The first phase splits the records into runs; and the second phase merges the runs until the table is sorted. We already know the benefits of using multiple buffers during the merge phase. It turns out that the split phase can also take advantage of additional buffers, as follows.

Suppose that k buffers are available. The split phase can read k blocks of the table at a time into the k buffers, use an internal sorting algorithm to sort them into a single k-block run, and then write those blocks to a temporary table. That is, instead of splitting the records into one-block-long runs, it splits the records into k-block-long runs. If k is large enough (in particular, if $k \geq \sqrt{B}$), then the split phase will produce no more than k initial runs; in this case, the preprocessing stage will not need to do any merging.

We summarize these ideas in the *multibuffer mergesort* algorithm of Figure 23-1.

```
        // The split phase, which uses $k$ buffers
    1.  Repeat until there are no more input records:
          a. Read $k$ blocks worth of input records into a new temporary table.
             Keep all $k$ of the buffers pinned.
          b. Use an internal sorting algorithm to sort these records.
          c. Unpin the buffers and write the blocks to disk.
          d. Add the temporary table to the run-list.

        // The merge phase, which uses $k+1$ buffers
    2.  Repeat until the run-list contains one temporary table:
             // Do an iteration
          a. Repeat until the run-list is empty:
               i.   Open a scan for $k$ of the temporary tables.
               ii.  Open a scan for a new temporary table.
               iii. Merge the $k$ scans into the new one.
               iv.  Add the new temporary table to list L.
          b. Add the contents of L to the run-list.
```

Figure 23-1: The Multibuffer Mergesort Algorithm

Step 1 of this algorithm produces *B/k* initial runs. The cost analysis of Section 22.4.4 tells us that the algorithm therefore requires $log_k(B/k)$ merge iterations. Since this expression is equivalent to (log_kB)-1, it follows that multibuffer mergesort requires one fewer merge iteration than basic mergesort (where the initial runs are of size 1). Put another way, multibuffer mergesort saves *2B* block accesses during the preprocessing stage.

The cost of multibuffer sorting a B-block table, using k buffers, is:
- *Preprocessing cost =* $2Blog_kB - 3B$ *+ the cost of its input*
- *Scanning cost = B*

We are now able to talk about how to choose the best value of *k*. The value of *k* determines the number of merge iterations. In particular, the number of iterations performed during preprocessing is equal to (log_kB)-2. It follows that:

- There will be 0 iterations when $k = \sqrt{B}$.
- There will be 1 iteration when $k = \sqrt[3]{B}$.
- There will be 2 iterations when $k = \sqrt[4]{B}$.

And so on.

This calculation should make intuitive sense to you. If $k = \sqrt{B}$, then the split phase will produce *k* runs of size *k*. These runs can be merged during the scanning phase, which means that no merge iterations are needed during preprocessing. And if $k = \sqrt[3]{B}$ then the split phase will produce k^2 runs of size k. One merge iteration will produce k runs (of size k^2), which can then be merged during the scanning phase.

For a concrete example, suppose that we need to sort a 4GB table. If blocks are 4K bytes, then the table contains about 1 million blocks. Figure 23-2 lists the number of buffers required to obtain a specific number of merge iterations during preprocessing.

# buffers	1,000	100	32	16	10	8	6	5	4	3	2
# iterations	0	1	2	3	4	5	6	7	8	11	18

Figure 23-2: The number of preprocessing iterations required to sort a 4GB table

At the lower end of this figure, note how adding just a few more buffers results in dramatic improvements: Two buffers require 18 iterations, but ten buffers bring it down to only 4 iterations. This tremendous difference in cost implies that it would be a very bad idea for the database system to sort this table using less than ten buffers.

The upper end of this figure illustrates how efficient sorting can be. It is quite possible that 1,000 buffers are available, or at least 100. The figure shows that with 1,000 buffers (or equivalently, 4MB of memory), it is possible to sort a 4GB table by performing 1,000 internal sorts during the preprocessing stage, followed by a single 1,000-way merge during the scanning phase. The total cost is three million block accesses: one million to read the unsorted blocks, one million to write to the temporary tables, and one million to read the temporary tables. This is a remarkably – and perhaps an unexpectedly – efficient way to sort the table.

This example also shows that for a given table size B, multibuffer mergesort can effectively use certain numbers of buffers, namely, $\sqrt{B}$, $\sqrt[3]{B}$, $\sqrt[4]{B}$, and so on. Figure 23-2 listed those values for B=1,000,000. What about other buffer values? What happens if we have, say, 500 buffers available? We know that 100 buffers result in 1 preprocessing merge iteration. Let’s see if those extra 400 buffers can be put to good use. With 500 buffers, the split phase will result in 2,000 runs of 500 blocks each. The first merge iteration will merge 500 runs at a time, resulting in 4 runs (of 250,000 blocks each). These runs can then be merged during the scanning phase. So in fact the extra 400 buffers don’t help us, because we still need the same number of iterations as 100 buffers.

This analysis can be expressed as the following rule:

If we use k buffers to sort a table that is B blocks long,
then k should be a root of B.

23.3 MultiBuffer Product

The basic implementation of the *product* operator is expensive. For example, consider the SimpleDB implementation of the query

$$\texttt{product}(T_1,\ T_2)$$

That implementation will examine all of T_2 for each record of T_1, using a single buffer to hold the records from T_2. That is, after it examines the last record of a T_2 block, it unpins the block and pins the next block of T_2. This unpinning allows the buffer manager to replace each T_2 block, which means that they all may need to be re-read from disk when the next record of T_1 is examined. In the worst case, each block of T_2 will be read as many times as there are records in T_1. If we assume that T_1 and T_2 are both 1,000-block tables containing 20 records per block, then the query will require 20,001,000 block accesses.

Suppose instead that the implementation did not unpin any blocks from T_2. The buffer manager would then be compelled to place each block of T_2 in its own buffer and not replace it. The blocks of T_2 will thus get read once from disk and remain in memory during the entire query. This scan would be exceptionally efficient, because it would read each block of T_1 once and each block of T_2 once.

Of course, this strategy will work only if there are enough buffers to hold all of T_2. What should we do if T_2 is too large? For example, suppose that T_2 is 1,000 blocks long, but only 500 buffers are available. The best thing to do is to process T_2 in two stages. First we read the first 500 blocks into the available buffers, and compute the product of T_1 with those blocks; then we compute the product of T_1 with the remaining 500 blocks of T_2. Analyzing block accesses, we see that the first stage requires reading T_1 once and the first half of T_2 once; and the second stage requires reading T_1 again and the second half of T_2 once. In total, T_1 gets read twice and T_2 gets read once.

The *multibuffer product* algorithm generalizes these ideas; see Figure 23-3.

Let T_1 and T_2 be the two input tables. Assume that T_2 is stored (as either a user-defined table or a materialized temporary table) and contains B_2 blocks.

1. Let $k = B_2/i$ for some integer i. That is, k is some fraction of B_2.
2. Treat T_2 as consisting of *i chunks* of k blocks each. For each chunk C:
   a) Read all of C's blocks into the k buffers.
   b) Take the product of T_1 and C.
   c) Unpin C's blocks.

Figure 23-3: The multibuffer product algorithm

In this algorithm, the blocks of T_1 will be read once for each chunk. Since there are B_2/k chunks, the number of block accesses in the product can be estimated by the formula:

$$B_2 + B_1*B_2/k$$

Note how the multibuffer product implementation treats T_1 and T_2 opposite from how they are treated by the basic product implmenetation of Chapter 17: In that chapter, T_2 is scanned multiple times, whereas here, T_1 is scanned multiple times.

Assume again that T_1 and T_2 are both 1,000-block tables. Figure 23-4 lists the block accesses required by the multibuffer product algorithm for various numbers of buffers.

# buffers	1,000	500	334	250	200	167	143	125	112	100
# chunks	1	2	3	4	5	6	7	8	9	10
# block accesses	2,000	3,000	4,000	5,000	6,000	7,000	8,000	9,000	10,000	11,000

Figure 23-4: The block accesses required to take the product of two 1,000-block tables

If 1,000 buffers are available, then T_2 can be processed in a single chunk, resulting in only 2,000 block accesses. On the other hand, if 250 buffers are available, then the multibuffer product algorithm would use 4 chunks of 250 blocks each; thus table T_1 would be scanned 4 times and T_2 would be scanned once, for a total of 5,000 block accesses. If only 100 buffers are available, then the algorithm would use 10 chunks, and thus 11,000 total block accesses. All of these values are much less than what the basic product implementation requires.

As with sorting, Figure 23-4 also demonstrates that not all values of *k* are useful. In this example, if 300 buffers are available, then the multibuffer product algorithm can only make use of 250 of them.

23.4 Implementing the Multibuffer Operations

23.4.1 Determining how many buffers to use

Each of our multibuffer algorithms chooses *k* buffers, but does not specify the exact value of *k*. The proper value of *k* is determined by the number of available buffers, the size of the input tables, and the operator involved. For *sort*, *k* is a root of the input table size; for *product*, *k* is a factor of the table size.

The goal is to choose *k* to be the largest root (or factor) that is less than the number of available buffers. The SimpleDB class *BufferNeeds* contains methods to calculate these values; its code appears in Figure 23-5.

```
public class BufferNeeds {

   public static int bestRoot(int size) {
      int avail = SimpleDB.bufferMgr().available();
      if (avail <= 1)
         return 1;
      int k = Integer.MAX_VALUE;
      double i = 1.0;
      while (k > avail) {
         i++;
         k = (int)Math.ceil(Math.pow(size, 1/i));
      }
      return k;
   }

   public static int bestFactor(int size) {
      int avail = SimpleDB.bufferMgr().available();
      if (avail <= 1)
         return 1;
      int k = size;
      double i = 1.0;
      while (k > avail) {
         i++;
         k = (int)Math.ceil(size / i);
      }
      return k;
   }
}
```

Figure 23-5: The code for the SimpleDB class *BufferNeeds*

The class contains the public static methods *bestRoot* and *bestFactor*. These two methods are almost identical. The input to each method is the size of the table. The methods first obtain the current number of available buffers *avail*, and then calculate the optimum number of buffers, either as the largest root or the largest factor that is less than *avail*. The method *bestRoot* initializes the variable *k* to *MAX_VALUE* in order to force the loop to be executed at least once (so that *k* cannot be more than $\sqrt{B}$).

Note that the methods in *BufferNeeds* do not actually reserve the buffers from the buffer manager. Instead, they simply determine how many buffers are currently available, and choose a value for *k* less than that. When the multibuffer algorithms attempt to pin those *k* blocks, some of the buffers may no longer be available. In that case, the algorithms will wait until the buffers become available again.

23.4.2 Implementing multibuffer sorting

In the SimpleDB class *SortPlan*, the methods *splitIntoRuns* and *mergeAllRuns* determine how many buffers to use. Currently, *splitIntoRuns* uses one buffer, and *mergeAllRuns* uses three buffers (two buffers for the input runs, and one buffer for the output run).

Consider first the method *splitIntoRuns*. This method does not actually know how large the sorted table will be, because the table has not yet been created. Thus the method must estimate the final size of the sorted table. It happens that the method *blocksAccessed* calculates exactly this estimate. Thus the method *splitIntoRuns* can execute the following code fragment:

```
int size = blocksAccessed();
int numbuffs = BufferNeeds.bestRoot(size);
```

Now consider the method *mergeAllRuns*. The best strategy is for the method to allocate *k* buffers, where *k* is a root of the number of initial runs. Thus we should revise *splitIntoRuns* so that it returns the number of runs created, and pass that value into *mergeAllRuns* (say, as the variable *numberOfInitialRuns*). The code for *mergeAllRuns* can then execute the following statement:

```
int numbuffs = BufferNeeds.bestRoot(numberOfInitialRuns);
```

The SimpleDB distribution code does not contain a version of *SortPlan* that performs multibuffer sorting. That task is left to Exercises 23.10-23.12.

Finally, note that code that uses *SortPlan*, such as *GroupByPlan* and *MergeJoinPlan*, cannot tell whether it is using the regular sorting algorithm or the multibuffer algorithm. Thus the code for those classes does not need to be changed. (However, there are some minor issues related to the number of buffers used by *MergeJoinPlan*; see Exercise 23.5.)

23.4.3 Implementing multibuffer product

In order to implement the multibuffer *product* algorithm, we need to implement the notion of a *chunk*. Recall that a chunk is a *k*-block portion of a materialized table, with the property that that all blocks of the chunk fit into the available buffers. The class *ChunkScan* implements chunks as a scan of records; see Figure 23-6.

```
public class ChunkScan implements Scan {
   private List<RecordPage> pages;
   private int startbnum, endbnum, current;
   private Schema sch;
   private RecordPage rp;

   public ChunkScan(TableInfo ti, int startbnum, int endbnum,
                                                Transaction tx) {
      pages = new ArrayList<RecordPage>();
      this.startbnum = startbnum;
      this.endbnum   = endbnum;
      this.sch = ti.schema();
      String filename = ti.fileName();
      for (int i=startbnum; i<=endbnum; i++) {
         Block blk = new Block(filename, i);
         pages.add(new RecordPage(blk, ti, tx));
      }
      beforeFirst();
   }

   public void beforeFirst() {
      moveToBlock(startbnum);
   }

   public boolean next() {
      while (true) {
         if (rp.next())
            return true;
         if (current == endbnum)
            return false;
         moveToBlock(current+1);
      }
   }

   public void close() {
      for (RecordPage r : pages)
         r.close();
   }

   public Constant getVal(String fldname) {
      if (sch.type(fldname) == INTEGER)
         return new IntConstant(rp.getInt(fldname));
      else
         return new StringConstant(rp.getString(fldname));
   }

   public int getInt(String fldname) {
      return rp.getInt(fldname);
   }

   public String getString(String fldname) {
      return rp.getString(fldname);
   }

   public boolean hasField(String fldname) {
      return sch.hasField(fldname);
   }

   private void moveToBlock(int blknum) {
      current = blknum;
      rp = pages.get(current - startbnum);
      rp.moveToId(-1);
   }
}
```

Figure 23-6: The code for the SimpleDB class *ChunkScan*

The *ChunkScan* constructor is given the metadata about the stored table, together with the block number of the first and last blocks of the chunk. The constructor opens record pages for each block in the chunk and stores them in a list. The scan also keeps track of a current record page; initially, the current page is the first page in the list. The *next* method moves to the next record in the current page. If the current page has no more records, then the next page in the list becomes current. Unlike table scans, moving between blocks in a chunk scan does not close the previous record page (which would unpin its buffer). Instead, the record pages in a chunk are unpinned only when the chunk itself is closed.

The class *MultiBufferProductPlan* implements the multibuffer product algorithm; its code appears in Figure 23-7.

```
public class MultiBufferProductPlan implements Plan {
   private Plan lhs, rhs;
   private Transaction tx;
   private Schema schema = new Schema();

   public MultiBufferProductPlan(Plan lhs, Plan rhs, Transaction tx) {
      this.lhs = lhs;
      this.rhs = rhs;
      this.tx = tx;
      schema.addAll(lhs.schema());
      schema.addAll(rhs.schema());
   }

   public Scan open() {
      TempTable tt = copyRecordsFrom(rhs);
      TableInfo ti = tt.getTableInfo();
      Scan leftscan = lhs.open();
      return new MultiBufferProductScan(leftscan, ti, tx);
   }

   public int blocksAccessed() {
      // this guesses at the # of chunks
      int avail = SimpleDB.bufferMgr().available();
      int size = new MaterializePlan(rhs,tx).blocksAccessed();
      int numchunks = size / avail;
      return rhs.blocksAccessed() +
             (lhs.blocksAccessed() * numchunks);
   }

   public int recordsOutput() {
      return lhs.recordsOutput() * rhs.recordsOutput();
   }

   public int distinctValues(String fldname) {
      if (lhs.schema().hasField(fldname))
         return lhs.distinctValues(fldname);
      else
         return rhs.distinctValues(fldname);
   }

   public Schema schema() {
      return schema;
   }

   private TempTable copyRecordsFrom(Plan p) {
      Scan   src = p.open();
      Schema sch = p.schema();
      TempTable tt = new TempTable(sch, tx);
      UpdateScan dest = (UpdateScan) tt.open();
      while (src.next()) {
         dest.insert();
         for (String fldname : sch.fields())
```

```
            dest.setVal(fldname, src.getVal(fldname));
      }
      src.close();
      dest.close();
      return tt;
   }
}
```

Figure 23-7: The code for the SimpleDB class *MultiBufferProductPlan*

The method *open* calls the local method *copyRecordsFrom* to materialize the right-side records into a temporary table. It then opens a scan on the left-side records, and passes the scan and the metadata about the temporary table into the *MultiBufferProductScan* constructor.

The method *blocksAccessed* needs to know the size of the materialized right-side table, so that it can calculate the number of chunks. Since this table does not exist until the plan is opened, the method needs to estimate the size. It does so by using the estimate provided by *MaterializePlan*. The code for the methods *recordsOutput* and *distinctValues* is the same as in *ProductPlan*, and is straightforward.

The code for *MultiBufferProductScan* appears in Figure 23-8.

```
public class MultiBufferProductScan implements Scan {
   private Scan lhsscan, rhsscan=null, prodscan;
   private TableInfo ti;
   private Transaction tx;
   private int chunksize, nextblknum, filesize;

   public MultiBufferProductScan(Scan lhsscan, TableInfo ti,
                                            Transaction tx) {
      this.lhsscan = lhsscan;
      this.ti = ti;
      this.tx = tx;
      filesize = tx.size(ti.fileName());
      chunksize = BufferNeeds.bestFactor(filesize);
      beforeFirst();
   }

   public void beforeFirst() {
      nextblknum = 0;
      useNextChunk();
   }

   public boolean next() {
      while (!prodscan.next())
         if (!useNextChunk())
            return false;
      return true;
   }

   public void close() {
      prodscan.close();
   }

   public Constant getVal(String fldname) {
      return prodscan.getVal(fldname);
   }

   public int getInt(String fldname) {
      return prodscan.getInt(fldname);
   }

   public String getString(String fldname) {
      return prodscan.getString(fldname);
   }

   public boolean hasField(String fldname) {
      return prodscan.hasField(fldname);
   }

   private boolean useNextChunk() {
      if (rhsscan != null)
         rhsscan.close();
      if (nextblknum >= filesize)
         return false;
      int end = nextblknum + chunksize - 1;
      if (end >= filesize)
         end = filesize - 1;
      rhsscan = new ChunkScan(ti, nextblknum, end, tx);
      lhsscan.beforeFirst();
      prodscan = new ProductScan(lhsscan, rhsscan);
      nextblknum = end + 1;
      return true;
   }
}
```

Figure 23-8: The code for the SimpleDB class *MultiBufferProductScan*

Its constructor determines the chunk size by calling *BufferNeeds.bestFactor* on the size of the materialized file. At this point, that file has just been materialized. The constructor can therefore call *tx.size* to determine the actual file size, and pass that into *bestFactor*.

The constructor then positions its left-side scan at the first record, opens a scan for the first chunk, and creates a *ProductScan* from these two scans. That is, the variable *prodscan* contains a basic product scan between the left-side scan and the current chunk. Most of the scan methods can be answered using this product scan. The exception is the method *next*.

The *next* method moves to the next record in the current product scan. If that scan has no more records, then the method closes that scan, creates a new product scan for the next chunk, and moves to its first record. The method returns *false* when there are no more chunks left to process.

23.5 Hash Joins

Section 22.6 examined the mergejoin algorithm. That algorithm sorts both of its input tables, which means that its cost is determined by the size of the larger input table.

In this section we consider a different join algorithm, called *hashjoin*. This algorithm has the property that its cost is determined by the size of the smaller input table. Thus this algorithm will be most efficient than mergejoin when the input tables are of very different sizes.

23.5.1 The *hashjoin* algorithm

The hashjoin algorithm appears in Figure 23-9.

```
Let $T_1$ and $T_2$ be the tables to be joined.

1. Choose a value $k$ that is less than the number of available buffers.
2. If the size of $T_2$ is no more than $k$ blocks, then:
   a) Join $T_1$ and $T_2$, using a multibuffer product followed by a selection
      on the join predicate.
   b) Return.

// Otherwise:
3. Choose a hash function that returns a value between 0 and $k$-1.
4. For the table $T_1$:
   a) Open a scan for $k$ temporary tables.
   b) For each record of $T_1$:
         i. Hash the record's join field, to get the hash value $h$.
         ii. Copy the record to the $h^{th}$ temporary table.
   b) Close the temporary table scans.
5. Repeat Step 4 for the table $T_2$.
6. For each $i$ between 0 and $k$-1:
   a) Let $V_i$ be the $i^{th}$ temporary table of $T_1$.
   b) Let $W_i$ be the $i^{th}$ temporary table of $T_2$.
   c) Recursively perform the hashjoin of $V_i$ and $W_i$.
```

Figure 23-9: The *hashjoin* algorithm

The algorithm is recursive, based on the size of T_2. If T_2 is small enough to fit in the available buffers, then we perform a multibuffer product of T_1 and T_2. Recall that multibuffer product is as fast as you can get when T_2 fits into memory, because each table will be scanned exactly once.

If T_2 is too large to fit into memory, then we use hashing to reduce its size. We create two sets of temporary tables: a set $\{V_0,...,V_{k-1}\}$ for T_1 and a set $\{W_0,...,W_{k-1}\}$ for T_2. These temporary tables act as buckets for the hash function. That is, we hash each T_1

record on its join field, and place it in the bucket associated with the hash value. Then we do the same thing for each T_2 record.

It should be clear that all records having the same join value will hash to the same bucket. Thus we can perform the join of T_1 and T_2 by individually joining V_i with W_i, for each *i*. Since each W_i will be smaller than T_2, we know that the recursion will eventually stop.

Note that each recursive call to the hashjoin algorithm must use a different hash function. The reason is that all of the records in a temporary table are there because they all hashed to the same value. A different hash function ensures that those records will be evenly distributed among the new temporary tables.

The algorithm also says to re-choose the value for *k* for each recursive call. We could also choose *k* once, and use it throughout all of the calls. Exercise 23.11 asks you to consider the tradeoffs involved in these two options.

We can improve the efficiency of the multibuffer product a little bit, by being careful how we search the blocks for matching records. Given a record of T_1, the algorithm needs to find the matching records from T_2. The strategy taken by multibuffer product is to simply search all of T_2. Although this search does not incur any additional disk accesses, it could certainly be made more efficient by means of appropriate internal data structures. For example, we could store references to the T_2 records in a hash table or binary search tree. (In fact, any implementation of the Java *Map* interface would work.) Given a T_1 record, the algorithm would look up its join value in the data structure, and find the references to the records of T_2 having this join value. Consequently, we avoid the need to search T_2.

23.5.2 An example of hashjoin

As a concrete example, let's use hashjoin to implement the join of the ENROLL and STUDENT tables, using the records from Figure 1-1. We make the following assumptions:

- The STUDENT table is on the right side of the join.
- Two STUDENT records fit in a block, and two ENROLL records fit in a block.
- We use three buckets; that is, *k*=3.
- Our hash function is *h(n)= n%3*.

The 9 STUDENT records fit into 5 blocks. Since *k*=3, we cannot fit all of STUDENT into memory at once, and so we hash. The resulting buckets appear in Figure 23-10.

[redacted]

Figure 23-10: Using hashjoin to join ENROLL with STUDENT

The student ID values 3, 6, and 9 have a hash value of 0. Thus the ENROLL records for those students are placed in V_0, and the STUDENT records for those students are placed in W_0. Similarly, the records for students 1, 4, and 7 are placed in V_1 and W_1, and the records for students 2, 5, and 8 are placed in V_2 and W_2. We now are able to recursively join each V_i table with its corresponding W_i table.

Since each W_i table has two blocks, they will each fit into memory; thus each of the three recursive joins can be performed as a multibuffer product. In particular, we join V_i with W_i by reading all of W_i into memory. We then scan V_i; for each record, we search W_i for any matching records.

23.5.3 Cost analysis

We now analyze the cost of using hashjoin to join T_1 with T_2. Suppose that the materialized records in T_1 require B_1 blocks, and that the records in T_2 require B_2 blocks. Choose k to be the n^{th} root of B_2; that is, $B_2 = k^n$. Then assuming that the records hash evenly, we can calculate the costs as follows:

The first round of hashing will produce k temporary tables; each of T_2's tables will be of size k^{n-1}. When we recursively hash these temporary tables, we will be left with k^2 temporary tables; each of T_2's tables will now be of size k^{n-2}. Continuing, we will eventually wind up with k^{n-1} temporary tables, with each of T_2's tables being of size k. These tables can then be joined (together with their corresponding tables from T_1) using multibuffer product.

Consequently, there will be *n*-1 rounds of hashing. The first round has cost B_1+B_2, plus the cost of reading their input. During subsequent rounds, each block of each temporary table will be read once, and written once; thus the cost for those rounds is 2(B_1+B_2). The multibuffer products occur during the scanning phase. Each block of the temporary tables will be read once, for a cost of B_1+B_2.

When we combine these values, we obtain the following formula:

The cost of using hashjoin to join tables of size B_1 and B_2, using k buffers, is:

- *Preprocessing cost = $(2B_1\log_k B_2 - 3B_1) + (2B_2\log_k B_2 - 3B_2)$ + the cost of the input*
- *Scanning cost = $B_1 + B_2$*

Amazingly enough, this cost is almost identical to the cost of a multibuffer mergejoin! There is one difference: in this formula, the argument to both of the logarithms is B_2; whereas in the formula for mergejoin, the argument to the first logarithm would be B_1. The reason for this difference is that in hashjoin, the number of rounds of hashing is determined only by T_2, whereas in mergejoin, the number of merge iterations during the sort phase is determined by both T_1 and T_2.

This difference explains the different performances of the two join algorithms. The mergejoin algorithm must sort both input tables before it can merge them. On the other hand, the hashjoin algorithm does not care how large T_1 is; it only needs to hash until T_2's buckets are small enough. The cost of a mergejoin is not affected by which table is on the left or right side. However, a hashjoin is more efficient when the smaller table is on the right.

A hashjoin is most efficient
when the smaller table is on the right side of the join.

If T_1 and T_2 are close in size, then it is probably better to use mergejoin, even though hashjoin has the same cost formula. The reason is that the hashjoin formula depends on the assumption that the records will hash evenly. In practice, of course, hashing rarely comes out evenly, which means that hashing may require more buffers and more iterations than the formula says. Mergejoin, on the other hand, has a much more predictable behavior.

23.6 Comparing the Join Algorithms

Given a multi-table SQL query, we have considered three ways to implement the join:

- Do a *mergejoin*.
- Do a *hashjoin*.
- Do an *indexjoin*.

In this section we investigate the relative benefits of each implementation, using the following example join query:

```
select SName, Grade from STUDENT, ENROLL where SId=StudentId
```

We assume that the tables have the sizes given in Figure 16-7, and that 200 buffers are available. We also assume that ENROLL has an index on *StudentId*.

Consider the mergejoin algorithm. This algorithm needs to sort both ENROLL and STUDENT before merging them. The ENROLL table has 50,000 blocks. The square root of 50,000 is 244, which is more than the number of available buffers. Thus we allocate the cube root, which is 37 buffers. The split phase will create 1352 runs, each of which is 37 blocks. A single merge iteration will result in 37 runs of size 1352 blocks. Thus preprocessing the ENROLL table requires two reads and two writes of the records, or 200,000 total block accesses. The STUDENT table has 4,500 blocks. The square root of 4,500 is 68, and 68 buffers are available. So we use 68 buffers to split the 4,500 STUDENT blocks into 68 runs of size 68. This splitting takes 9,000 block accesses, and is all the preprocessing that is needed. Merging the two sorted tables requires another 54,500 block accesses, for a total cost of 263,500 block accesses.

Consider now the hashjoin algorithm. This algorithm is most efficient when the smallest table is on the right; thus ENROLL will be the left-side table and STUDENT will be the right-side table. We can use 68 buffers to hash STUDENT into 68 buckets, each of which will contain about 68 blocks. Similarly, we use the same 68 buffers to hash ENROLL into 68 buckets, each of which will contain about 736 blocks. We then recursively join the corresponding buckets. Each of these sub-joins can be performed using multibuffer product. Thus we allocate 68 buffers to hold the entire STUDENT bucket, and we allocate another buffer for a sequential scan through the ENROLL bucket. Each bucket gets scanned once. Summing the costs, we see that we read ENROLL and STUDENT once, write the buckets once, and read them once, for a total of 163,500 block accesses.

The *indexjoin* implementation scans through the STUDENT table; for each STUDENT record, it uses the record's *SId* value to search the index, and then looks up the matching ENROLL records. Thus the STUDENT table will be accessed once (for 4,500 block accesses), and the ENROLL table will be accessed once for each matching record. However, note that every ENROLL record matches some STUDENT record, which means that the ENROLL table will potentially require 1,500,000 block accesses. The query therefore requires 1,504,500 block accesses.

Our analysis shows that under these assumptions, hashjoin is the fastest, followed by mergejoin and then indexjoin. The reason why hashjoin is so efficient is that one of the tables (i.e. STUDENT) is reasonably small compared to the number of available buffers, and the other (i.e. ENROLL) is much larger. Suppose instead that 1,000 buffers were available. Then mergejoin would be able to sort ENROLL without any merge iterations, and thus the total cost would be 163,500 block accesses, the same as hashjoin. The indexjoin algorithm is by far the least efficient implementation for this query. The reason is that indexes are not useful when there are many matching data records, and in this query every ENROLL record matches.

Now let's consider a variation of this query that has an additional selection on *GradYear*:

```
select SName, Grade
from STUDENT, ENROLL
where SId=StudentId and GradYear=2005
```

Consider first the mergejoin implementation. There are only 900 relevant STUDENT records, which fit into 90 blocks. Thus it is possible to sort the STUDENT records by reading them into 90 buffers and using an internal sort algorithm to sort them. Thus only 4,500 block accesses are needed. But the cost of processing ENROLL is unchanged, so the query would require a total of 204,500 block accesses, only a slight improvement over mergejoin on the original query.

The hashjoin implementation would recognize that the 90 blocks of STUDENT records will fit directly into 90 buffers, with no hashing required. Thus the join can be performed by a single scan of both tables, which is 54,500 block accesses.

The indexjoin implementation would read all 4,500 STUDENT records to find the 900 students from 2005. These records will match with 1/50th (or 50,000) of the ENROLL records, resulting in about 50,000 block accesses of ENROLL, or 54,500 total block accesses.

We therefore see that hashjoin and indexjoin are comparable, but mergejoin is significantly worse. The reason is that mergejoin is forced to preprocess both tables, even though one is considerably smaller.

For a final example, we modify the above query so that there is even a more restrictive selection on STUDENT:

```
select SName, Grade from STUDENT, ENROLL
where SId=StudentId and SId=3
```

Now the output table consists of the 34 records corresponding to the enrollments for this single student. In this case, indexjoin will be the most efficient. It scans the entire 4,500 blocks of STUDENT, traverses the index, and looks up the 34 ENROLL records, for a total of about 4,534 block accesses (not counting index traversal costs). The hashjoin implementation has the same cost as before. It will need to scan STUDENT once (to materialize the single record) and ENROLL once (to find all of the matching records), for a total of 54,500 block accesses. And mergejoin will have to preprocess ENROLL and STUDENT the same as before, for a total of 204,500 block accesses.

This analysis demonstrates that mergejoin is most efficient when both of its input tables are relatively the same size. Hashjoin is often better when at the input tables are of disparate sizes. And indexjoin is better when the number of output records is small.

23.7 Chapter Summary

- Non-materialized scans are very frugal when it comes to buffer usage. In particular:
  - A table scan uses exactly one buffer.
  - Scans for *select*, *project*, and *product* use no additional buffers.
  - A static hash or B-tree index requires one additional buffer (for queries).

- Materialized scans, on the other hand, can make good use of large numbers of buffers. In particular, a scan for *mergesort* requires *k*+1 additional buffers, where *k* is the number of auxiliary tables used.

- The *mergesort* algorithm can take advantage of multiple buffers when it creates the initial runs. It chooses $k = {}^{n}\sqrt{B}$, where *B* is the size of the input table and *n* is the smallest integer such that *k* is less than the number of available buffers. The resulting algorithm is called *multibuffer mergesort*, and is as follows:
  - Allocate *k* buffers from the buffer manager.
  - Read *k* blocks of the table at a time into the *k* buffers, and use an internal sorting algorithm to sort them into a *k*-block run.
  - Perform merge iterations on the resulting runs using *k* temporary tables, until there are no more than *k* runs remaining. Since the splitting phase results in *B/k* runs, there will be *n*-2 merge iterations.
  - Merge the final *k* runs during the scanning phase.

- The only buffer allocations useful for the multibufer mergesort algorithm are those that are a root of *B*.

- The *multibuffer product* algorithm is an efficient implementation of the *product* operator, and works as follows:
  1. Materialize the RHS table as temporary table T_2. Let B_2 be the number of blocks in T_2.
  2. Let *i* be the smallest number such that B_2/i is less than the number of available buffers. Reserve $k= B_2/i$ buffers from the buffer manager.
  3. Treat T_2 as *i chunks* of *k* blocks each. For each chunk *C*:
     a. Read all of *C*'s blocks into the *k* buffers.
     b. Take the product of T_1 and *C*.
     c. Unpin *C*'s blocks.

  That is, T_1's blocks will be read once for each chunk, leading to the following formula:

  ```
  The number of blocks in product(T₁, T₂) = B₂ + B₁*B₂/k
  ```

- The only values of *k* that are useful for the multibuffer product algorithm are ones that create equal-sized chunks. That is, useful values of *k* are factors of B_2.

- The *hashjoin* algorithm works as follows:
  1. Choose a value *k* that is less than the number of available buffers.
  2. If T_2 fits into *k* buffers, use a multibuffer product to join T_1 and T_2.

3. Otherwise, hash T_1 and T_2, using *k* temporary tables each.
4. Recursively perform hashjoin on the corresponding hashed buckets.

23.8 Suggested Reading

The article [Shapiro 1986] describes and analyzes several join algorithms and their buffer requirements. The article [Yu and Cornell 1993] considers the cost-effectiveness of buffer usage. It argues that buffers are a valuable global resource, and that instead of allocating as many buffers as it can (which is what SimpleDB does), a query should allocate the number of buffers that will be most cost-effective for the entire system. The article gives an algorithm that can be used to determine the optimal buffer allocation.

23.9 Exercises

CONCEPTUAL EXERCISES

23.1 Suppose that a database system contains so many buffers that they never all are pinned at the same time. Is this just a waste of physical memory, or is there an advantage to having an excess number of buffers?

23.2 Large amounts of RAM are becoming increasingly cheap. Suppose that a database system has more buffers than there are blocks in the database. Can all of the buffers be used effectively?

23.3 Suppose that the database system contains enough buffers to hold every block of the database. Such a system is called a *main-memory* database system, because it can read the entire database into buffers once, and then execute queries without any additional block accesses.
a) Does any component of the database system become unnecessary in this case?
b) Should the functioning of any component be significantly different?
c) The query plan estimation functions certainly need changing, because it no longer makes sense to evaluate queries based on the number of block accesses. Suggest a better function, which would more accurately model the cost of evaluating a query.

23.4 Consider the description of multibuffer sorting in Section 23.4.2, which suggests that the methods *splitIntoRuns* and *mergeAllRuns* should each allocate their buffers separately.
a) Another option would be for the *open* method to determine a value for *numbuffs* and pass it into both methods. Explain why this is a less desirable option.
a) Yet another option would be to allocate the buffers in the *SortPlan* constructor. Explain why this is an even worse option.

23.5 Suppose that the class *SortPlan* has been revised to implement the multibuffer sorting algorithm of Figure 23-2, and consider the first mergejoin example in Section 23.6.
a) How many buffers are used in the scanning phase of that mergejoin scan?

b) Suppose that only 100 buffers were available (instead of 200). Suppose that buffers were allocated for ENROLL before STUDENT. How would they be allocated?
c) Suppose that only 100 buffers were available (instead of 200). Suppose that buffers were allocated for STUDENT before ENROLL. How would they be allocated?
d) Another option would be to fully materialize either of the sorted tables, before joining them. Calculate the cost of this option.

23.6 Consider the example of multibuffer product in Section 23.3, which took the product of two 1,000-block tables. Suppose that only one buffer were available for table T_2; that is, suppose that we choose *k*=1.
a) Calculate the number of block accesses required to take the product.
b) This number is significantly less than the block accesses required by the basic product algorithm of Chapter 17, even though it uses the same number of buffers. Explain why.

23.7 The multibuffer product algorithm requires that the RHS table be materialized (so that it can be chunked). Although the LHS table need not be materialized, the query may be more efficient if it is materialized. Explain why, and give an example.

23.8 Consider the following algorithm for implementing the *groupby* operator:

```
1. Create and open k temporary tables.
2. For each input record:
   a) Hash the record on its grouping fields.
   b) Copy the record to the corresponding temporary table.
3. Close the temporary tables.
4. For each temporary table:
      Perform the sort-based groupby algorithm on that table.
```

a) Explain why this algorithm works.
b) Calculate the preprocessing and scanning costs of this algorithm.
c) Explain why this algorithm is in general not as good as the sort-based groupby algorithm of Figure 22-12.
d) Explain why this algorithm might be useful in a parallel-processing environment.

23.9 Rewrite the hashjoin algorithm of Figure 23-9 so that it is nonrecursive. Make sure that all of the hashing is performed during the preprocessing stage, and that the merging is performed during the scanning stage.

23.10 The hashjoin algorithm of Figure 23-9 uses the same value of *k* to hash the records of both T_1 and T_2. Explain why using different values of *k* will not work.

23.11 The hashjoin algorithm of Figure 23-9 re-chooses the value of *k* each time it is called.

a) Explain why it would also be correct to choose the value of *k* once, and pass it into each recursive call.
b) Analyze the tradeoffs of these two possibilities. Which do you prefer?

23.12 Suppose we revise the hashjoin algorithm of Figure 23-9 so that step 6 uses mergejoin to join the individual buckets, instead of calling hashjoin recursively. Give a cost analysis of this algorithm, and compare the block accesses to the original hashjoin algorithm.

23.13 Suppose that the STUDENT table has indexes on *SId* and *MajorId*. For each of the following SQL queries, use the statistics of Figure 16-7 to calculate the cost of implementations that use mergejoin, hashjoin, or indexjoin.

a)
```
select s.SName, d.DName from STUDENT s, DEPT d
where  s.MajorId=d.DId
```

b)
```
select s.SName, d.DName from STUDENT s, DEPT d
where  s.MajorId=d.DId and s.GradYear=2009
```

c)
```
select d.DName from STUDENT s, DEPT d
where  s.MajorId=d.DId and s.SId=1
```

d)
```
select s.SName from STUDENT s, ENROLL e
where  s.SId=e.StudentId and e.Grade='F'.
```

e)
```
select e.EId from STUDENT s, ENROLL e
where  s.SId=e.StudentId and s.SId=1.
```

PROGRAMMING EXERCISES

23.14 The SimpleDB class *BufferNeeds* does not reserve buffers from the buffer manager.
a) List some possible problems that could occur in SimpleDB that would be alleviated if the buffers were actually reserved. Are there any advantages to not reserving buffers?
b) Redesign the SimpleDB buffer manager so that it allows transactions to reserve buffers. (Be sure to consider the case where transaction T_1 pins block b to a reserved buffer, and then transaction T_2 wants to pin b. What should you do?)
c) Implement your design, and modify *BufferNeeds* appropriately.

23.15 In Exercise 22.10 you modified the class *SortPlan* so that it constructs initial runs that are one block long. Modify the code so that it constructs initial runs that are *k* blocks long, as discussed in Section 23.4.2.

23.16 In Exercise 22.11 you modified the class *SortPlan* to use a one-block long staging area for computing the initial runs. Modify the code so that it uses a *k*-block long staging area.

23.17 In Exercise 22.13 you modified the class *SortPlan* to merge *k* runs at a time, where the value of *k* was passed into the constructor. Modify the code so that the value of *k* is determined by the number of initial runs, as discussed in Section 23.4.2.

23.18 The multibuffer product algorithm is usually most efficient when its smallest input table is on the right side.
a) Explain why.
b) Give an example where the algorithm is more efficient when the smallest input table is on the left side. (Hint: Don't forget about the cost of materializing the table.)
c) Revise the code for *MultiBufferProductPlan* so that it always chooses the smaller input table to be on the right side of the scan.

23.19 Revise the code for *MultiBufferProductPlan* so that it doesn't materialize its right-side table unnecessarily.

23.20 Write SimpleDB code to implement the hashjoin algorithm.

24

QUERY OPTIMIZATION
and the package simpledb.opt

The basic planner of Chapter 19 uses a simple algorithm to create its plans. Unfortunately, those plans often entail significantly more block accesses than they need to, for two basic reasons:

- The operations are performed in a suboptimal order.
- The operations are implemented using the simple pipelined implementations of Chapter 17, and do not take advantage of the indexed, materialized, or multibuffer implementations of Chapters 21-23.

In this chapter we consider how the planner can address these problems and generate efficient plans; this task is called *query optimization*. The most efficient plan for a query can be several orders of magnitude faster than a naïve plan, which is the difference between a database system that can respond to queries in a reasonable amount of time and a database system that is completely unusable. A good query optimization strategy is therefore a vital part of every commercial database system.

24.1 Equivalent Query Trees

Two tables are equivalent if an SQL query cannot tell them apart.

In other words, the two tables must contain exactly the same records, although not necessarily in the same order.

Two queries are equivalent if their output tables are always equivalent, regardless of the contents of the database.

In this section we consider equivalences between relational algebra queries. Since these queries can be expressed as trees, an equivalence between two queries can often be thought of as a transformation between their trees. The following subsections consider these transformations.

24.1.1 Rearranging products

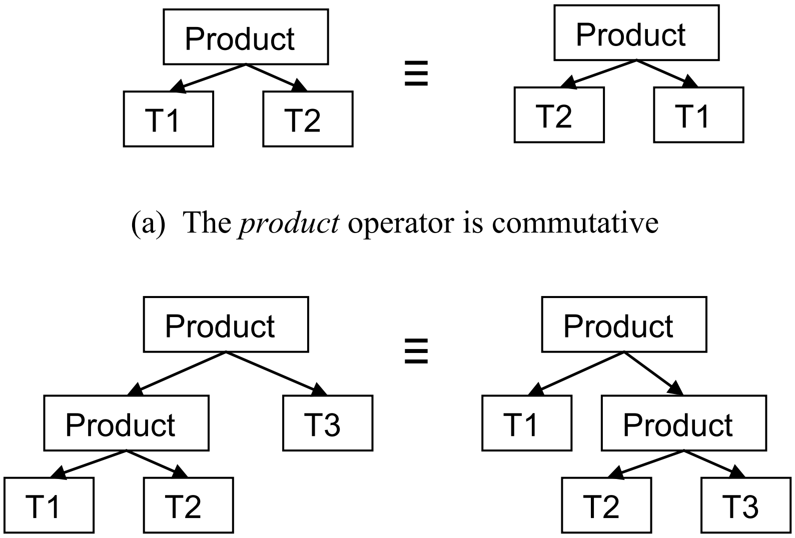
Let T_1 and T_2 be two tables. The product of T_1 and T_2 is the table containing all combinations of records from T_1 and T_2. That is, whenever there are records r_1 in T_1 and r_2 in T_2, then the combined record (r_1, r_2) is in the output table. Note that this combined record is essentially the same as (r_2, r_1), since the order in which fields appear in a record is irrelevant. But since (r_2, r_1) is the record produced by the product of T_2 and T_1, the product operator must be commutative. That is:

```
product(T₁, T₂)  ≡  product(T₂, T₁)
```

A similar argument (see Exercise 24.1) can show that the *product* operator is associative. That is:

```
product(product(T₁, T₂), T₃)  ≡  product(T₁, product(T₂, T₃))
```

In terms of query trees, the first equivalence swaps the left and right children of a *product* node. The second equivalence applies when two *product* nodes are next to each other; in that case, the inner *product* node moves from being the left child of the outer *product* node to being its right child; the ordering of the other child nodes stays the same. Figure 24-1(a) illustrates the first equivalence, and Figure 24-1(b) the second.



(a) The *product* operator is commutative

(b) The *product* operator is associative

Figure 24-1: Equivalences involving the *product* operator

These two equivalences can be used repeatedly to transform trees of *product* nodes. For example, consider Figure 24-2, which consists of two trees corresponding to the query

```
select SName
from STUDENT, ENROLL, SECTION, COURSE, DEPT
```

The tree from Figure 24-2(a) is created by the basic planner. Two steps are required to transform this tree into the tree of Figure 24-2(b). The first step applies the commutative rule to the *product* node above SECTION; the second step applies the associative rule to the *product* node above DEPT.

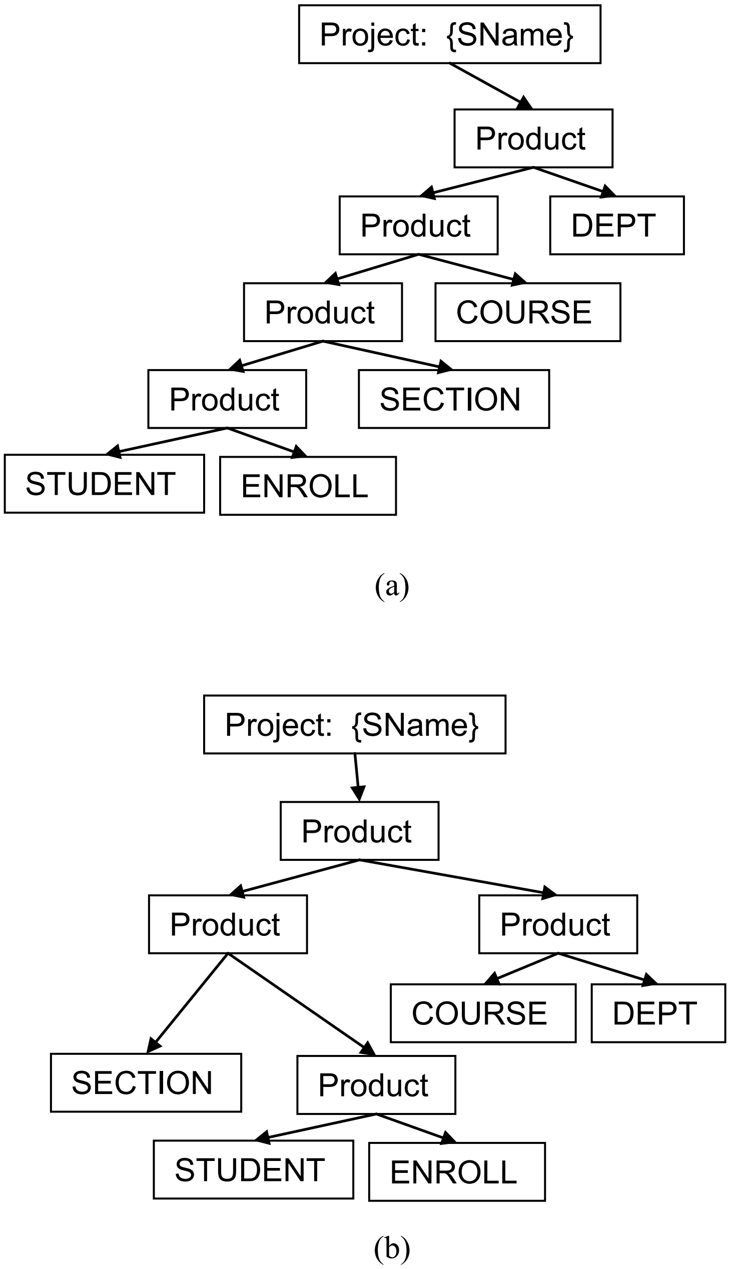


Figure 24-2: Rearranging *product* nodes to produce an equivalent query tree

In fact, it can be shown (see Exercise 24.3) that these two rules can be used to transform any tree of *product* nodes into any other tree having the same nodes. That is, *product* operations can be performed in any order.

24.1.2 Splitting selections

Suppose that predicate p is the conjunction of two predicates p_1 and p_2. Instead of using p directly, we can find the records satisfying p in two steps: First we find the records satisfying p_1, and then from that set we find the records satisfying p_2. In other words, the following equivalence holds:

$$\texttt{select}(T,\ p_1 \texttt{ and } p_2) \ \equiv\ \texttt{select}(\texttt{select}(T,\ p_1),\ p_2)$$

In terms of query trees, this equivalence replaces a single *select* node by a pair of *select* nodes; see Figure 24-3.

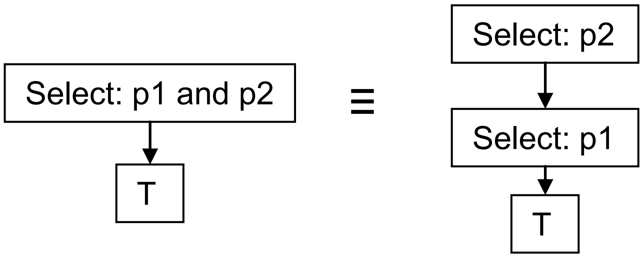


Figure 24-3: Splitting a *select* node

By applying this equivalence repeatedly, it is possible to replace the single *select* node in a query tree by several *select* nodes, one for each conjunct in the predicate. Moreover, since the conjuncts of a predicate can be arbitrarily rearranged, these *select* nodes can appear in any order.

The ability to split a *select* node is enormously useful for query optimization, because each of the "smaller" *select* nodes can be placed independently at its optimum location in the query tree. Consequently, the query optimizer strives to split predicates into as many conjuncts as possible. It does so by transforming each predicate into *conjunctive normal form* (or CNF).

A predicate is in CNF if it is a conjunction of sub-predicates,
none of which contain an AND operator.

The AND operators in a CNF predicate will always be outermost. For example, consider the following SQL query:

```
select SName
from STUDENT
where (MajorId=10 and SId=3) or (GradYear=2004)
```

As written, the where-clause predicate is not in CNF, because the AND operator is inside the OR operator. However, it is always possible to use DeMorgan's laws to make the AND operator be outermost. In this case, we have the equivalent query:

```
select SName
from STUDENT
where (MajorId=10 or GradYear=2004) and (SID=3 or GradYear=2004)
```

The predicate of this query has two conjuncts, which can now be split.

24.1.3 Moving selections within the tree

The following query retrieves the name of every student majoring in math:

```
select SName
from STUDENT, DEPT
where DName = 'math' and MajorId = DId
```

Its where-clause predicate is in CNF, and contains two conjuncts. Figure 24-4(a) depicts the query tree created by the basic planner, modified so that there are two *select* nodes.

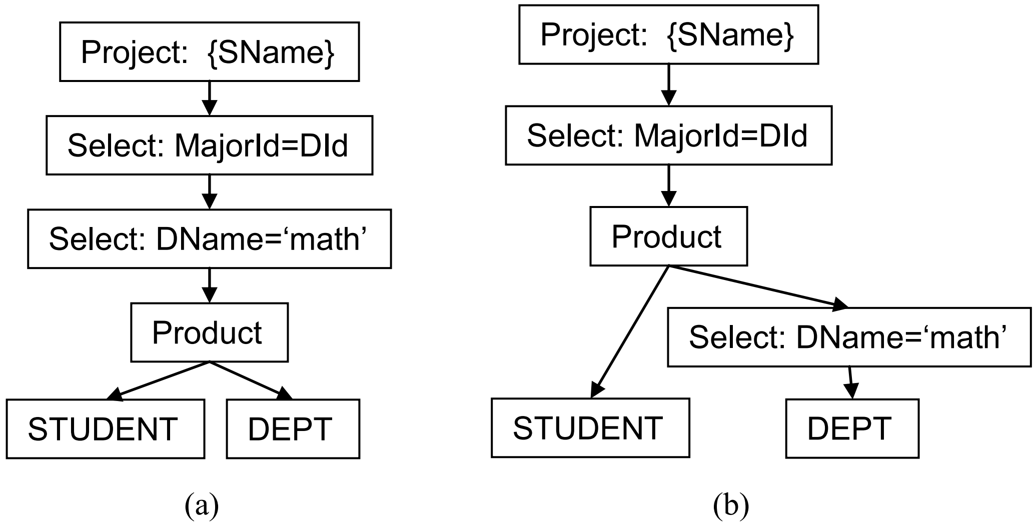


Figure 24-4: Pushing a *select* node down the query tree

Consider first the selection on *DName*. The *product* node below it outputs all combinations of STUDENT and DEPT records; the *select* node then retains only those combinations in which *DName* has the value 'math'. This is exactly the same set of records you would get if you first selected the math-department record from DEPT, and took all combinations of STUDENT records with that record. In other words, since the selection applies only to the DEPT table, it is possible to "push" the selection inside the product, giving the equivalent tree depicted in Figure 24-4(b).

Now consider the join predicate *MajorId = DId*. It is not possible push this selection inside the product, because the predicate mentions fields from both STUDENT and DEPT. For example, pushing the selection above STUDENT would produce a meaningless query, because the selection would reference a field that is not in STUDENT.

This discussion can be generalized as follows. If predicate p refers only to fields of T_1, then the following equivalence holds:

$$\texttt{select(product(}T_1\texttt{, }T_2\texttt{), p)} \equiv \texttt{product(select(}T_1\texttt{, p), }T_2\texttt{)}$$

This equivalence is depicted in Figure 24-5.

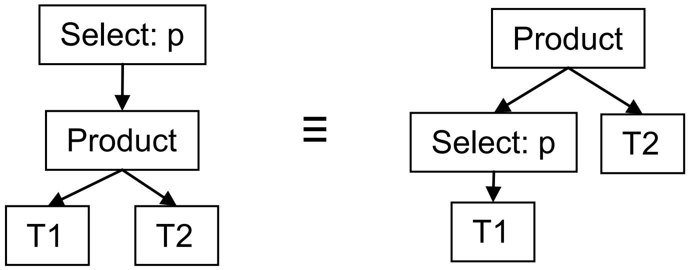
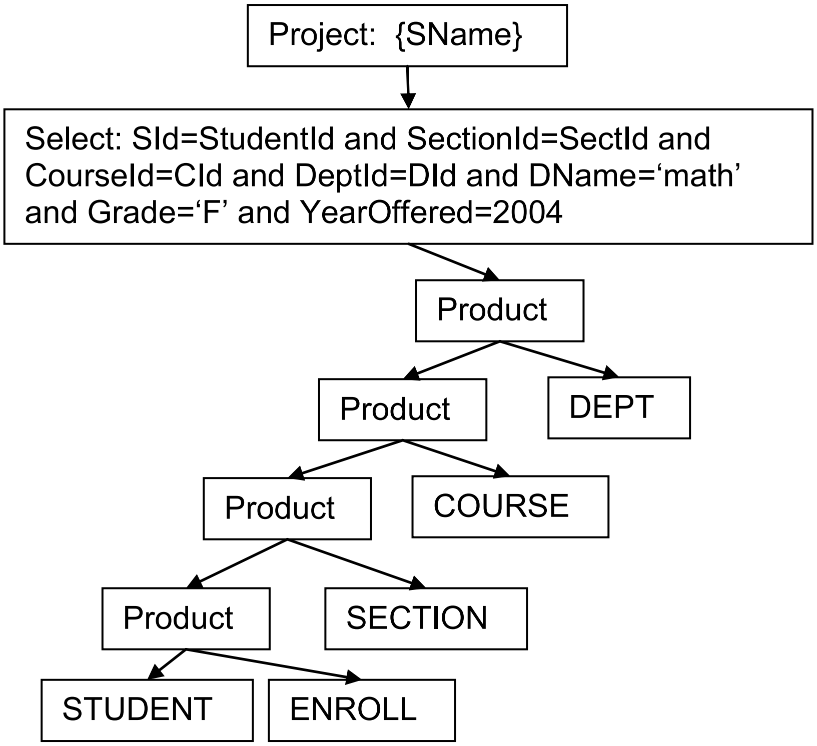


Figure 24-5: Pushing a *select* node inside of a *product*

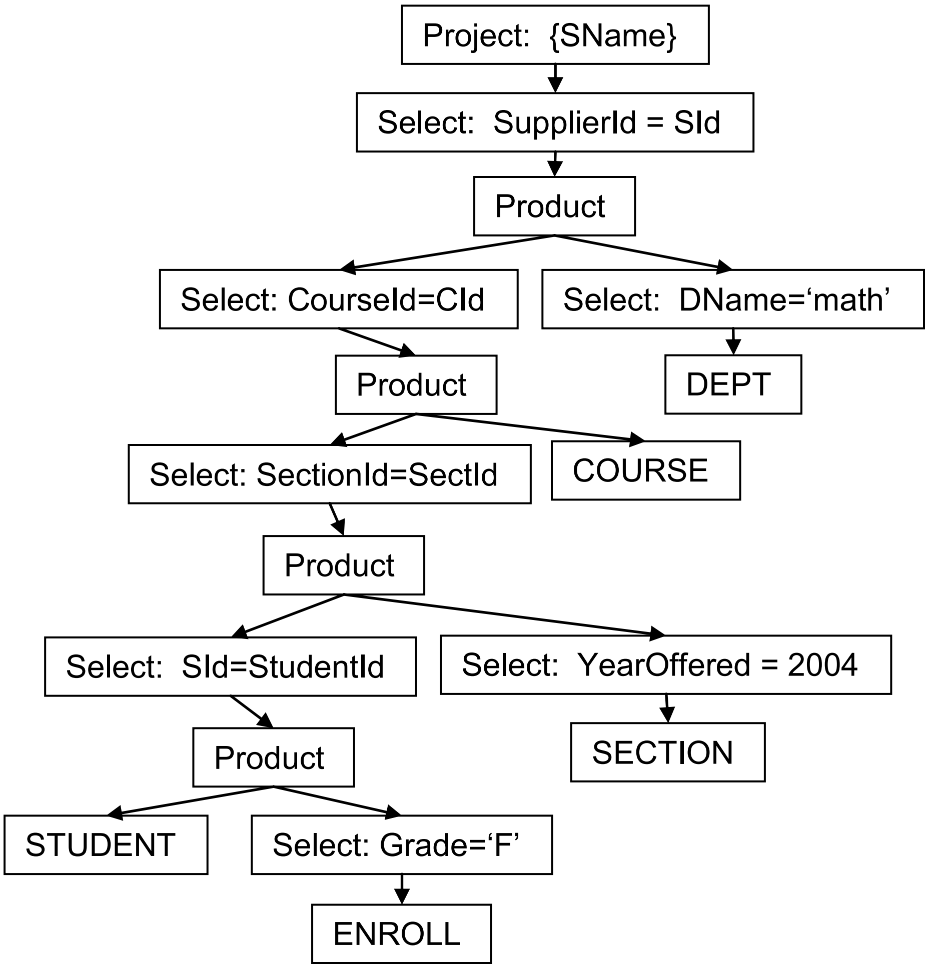
This equivalence can be applied repeatedly to a *select* node, pushing it down the query tree as far as possible. For example, consider Figure 24-6. The query of part (a) returns the name of those students who failed a math course in 2004. Parts (b) and (c) depict two equivalent trees for this query. Figure 24-6(b) depicts the query tree created by the basic planner. Figure 24-6(c) depicts the query tree resulting from splitting the select node and pushing the smaller select nodes down the tree.

```
select SName
from STUDENT, ENROLL, SECTION, COURSE, DEPT
where SId=StudentId and SectionId=SectId and CourseId=CId
and DeptId=DId and DName='math' and Grade='F' and YearOffered=2004
```

(a) The SQL query


Project: {SName}
Select: SId=StudentId and SectionId=SectId and CourseId=CId and DeptId=DId and DName='math' and Grade='F' and YearOffered=2004
Product
Product
DEPT
Product
COURSE
Product
SECTION
STUDENT
ENROLL

(b) The query tree created by the basic planner



(c) The query tree resulting from pushing *select* nodes

Figure 24-6: Pushing selections down a query tree

The equivalence of Figure 24-5 can also be applied in reverse, moving a *select* node up the tree past one or more *product* nodes. Moreover, it is easily shown that a *select* node can always be moved past another *select* node in either direction, and that a *select* node can be moved past a *project* or *groupby* node whenever it is meaningful to do so (see Exercise 24.4). It therefore follows that a *select* node can be placed anywhere in the query tree, provided that its predicate only mentions fields of the underlying subtree.

24.1.4 Identifying join operators

Recall that the *join* operator is defined in terms of the operators *select* and *product*:

```
join(T1, T2, p)  ≡  select(product(T1, T2), p)
```

This equivalence asserts that it is always possible to transform a pair of *select-product* nodes into a single *join* node. For example, Figure 24-7 depicts the result of this transformation on the tree of Figure 24-6(c).

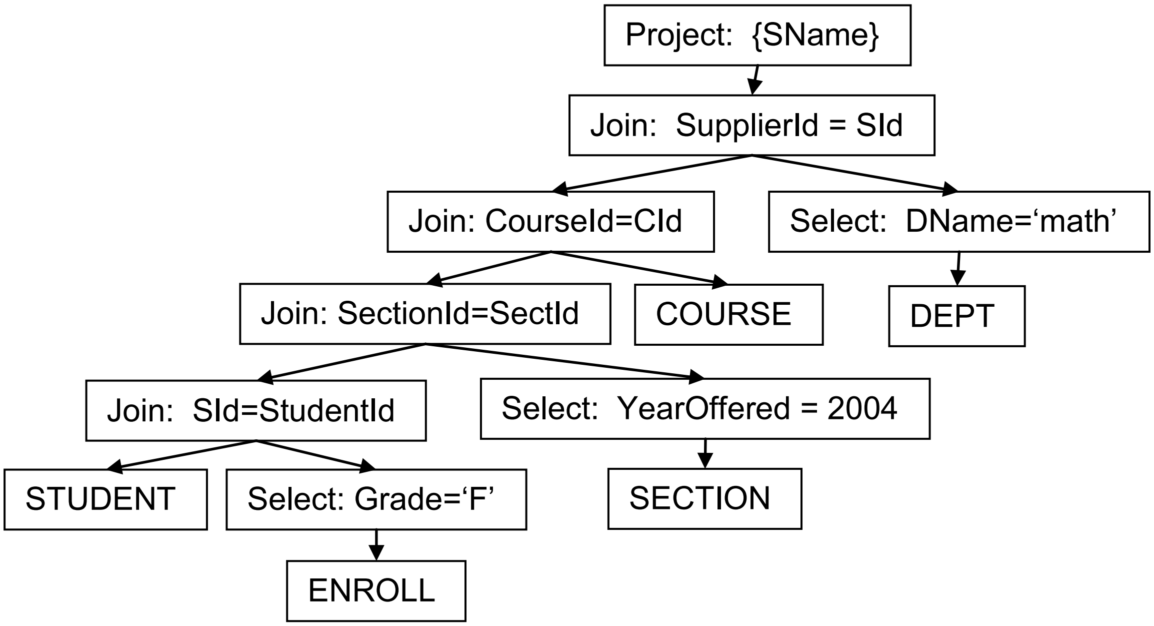


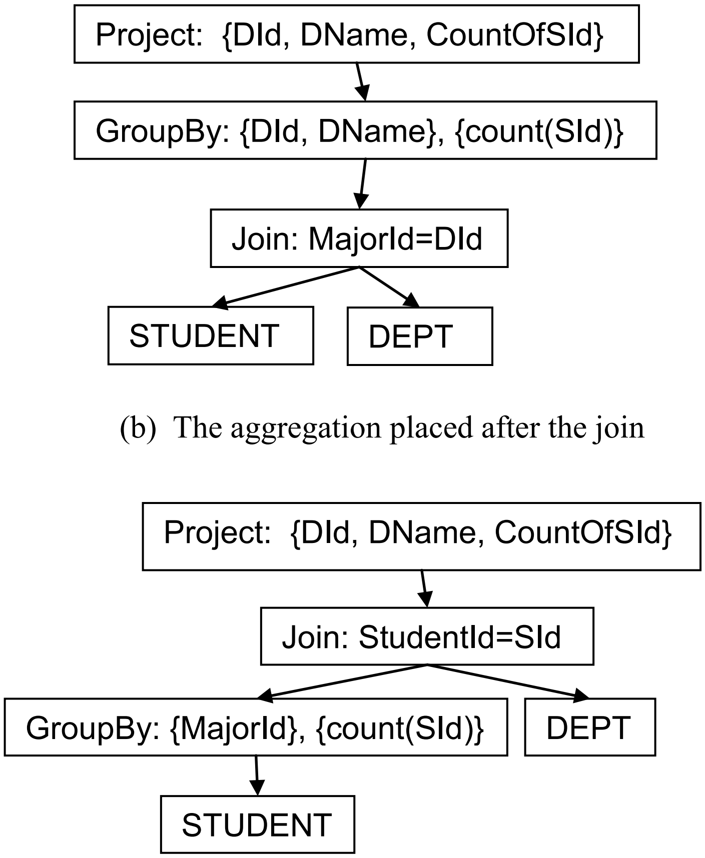
Figure 24-7: Replacing the *select-product* nodes in Figure 24-6(c) with *join* nodes

24.1.5 Moving an aggregation inside a join

Consider an SQL query that involves both a join and an aggregation, such as the query of Figure 24-8(a). That query retrieves, for each department that has at least one student major, the department's ID, name, and the number of students majoring in that department. The *group by* clause contains the fields *DId* and *DName*. Note that the field *DId* determines the groups. The field *DName* does not change the groups; it only appears in the *group by* clause so that it can be part of the output, as we discussed in Section 4.3.6.

```
select DId, DName, Count(SId)
from STUDENT, DEPT
where MajorId = DId
group by DId, DName
```

(a) The SQL query


Project: {DId, DName, CountOfSId}
GroupBy: {DId, DName}, {count(SId)}
Join: MajorId=DId
STUDENT
DEPT
Project: {DId, DName, CountOfSId}
Join: StudentId=SId
GroupBy: {MajorId}, {count(SId)}
DEPT
STUDENT

(b) The aggregation placed after the join

(c) Pushing the aggregation inside the join

Figure 24-8: Transforming a query tree having a *groupby* node

Figure 24-8(b) depicts the query tree created by the basic planning algorithm. Note that the *groupby* node has the following properties – its grouping fields all come from the DEPT table, and include the key; and its aggregation functions only mention fields from STUDENT. These properties make it possible to push the *groupby* node inside the join, as in Figure 24-8(c). Note how the new *groupby* node now has the foreign key *MajorId* as its grouping field.

This example can be generalized as the equivalence of Figure 24-9. This equivalence assumes the following conditions:

- The join is a relationship join. That is, the join predicate equates the foreign key of table T1 with the key of table T2.
- The grouping fields in the *groupby* node contain only fields from T2, and include the key.
- The aggregation functions mention only fields from T1.

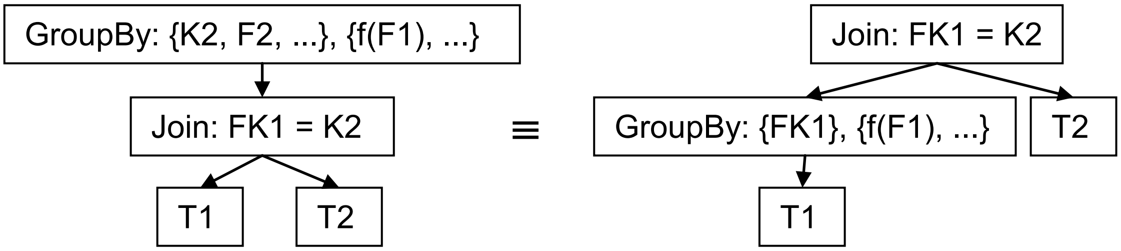

GroupBy: {K2, F2, ...}, {f(F1), ...}
Join: FK1 = K2
T1
T2
≡
Join: FK1 = K2
GroupBy: {FK1}, {f(F1), ...}
T2
T1

Figure 24-9: Pushing a *groupby* node inside of a *join*

The first two bullet points ensure that the foreign key field is sufficient to determine the groups. The other grouping fields can thus be omitted from the *groupby* node of the transformed tree, and will show up in the output via the join.

24.1.6 Adding projections

A *project* node can be added on top of any node in a query tree, provided that its projection list contains all fields mentioned in the ancestors of the node. This transformation is typically used to reduce the size of the inputs to the nodes in a query tree.

For example, Figure 24-10 depicts the query tree of Figure 24-7, where we have added *project* nodes to eliminate fields as early as possible.

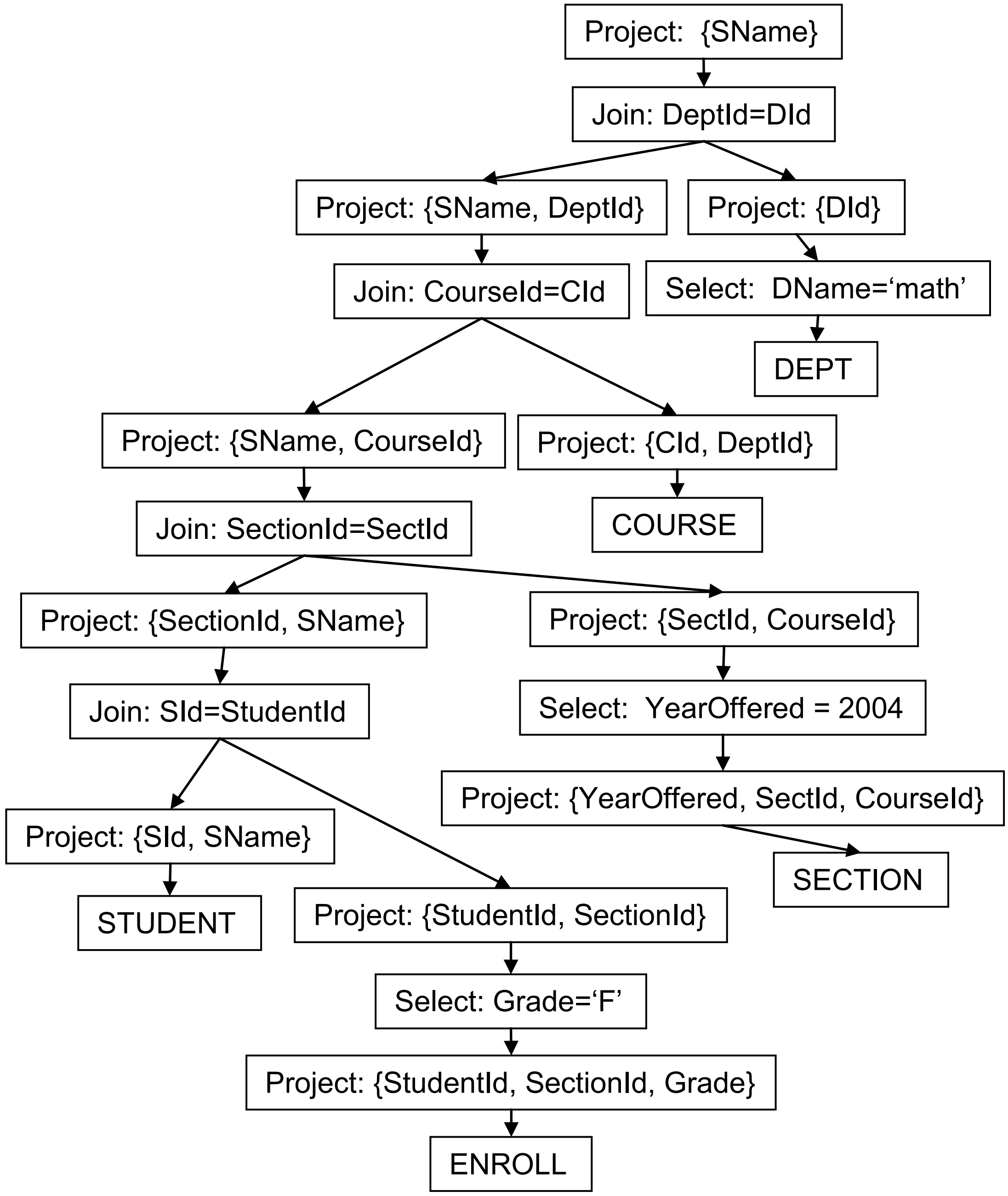


Figure 24-10: Adding projections to the query tree of Figure 24-6(c)

24.2 The Need for Query Optimization

Given an SQL query, the planner must choose an appropriate plan for it. We can divide this plan-generation activity into two steps:

- The planner chooses a relational algebra query tree corresponding to the query.
- The planner chooses an implementation for each node in the query tree.

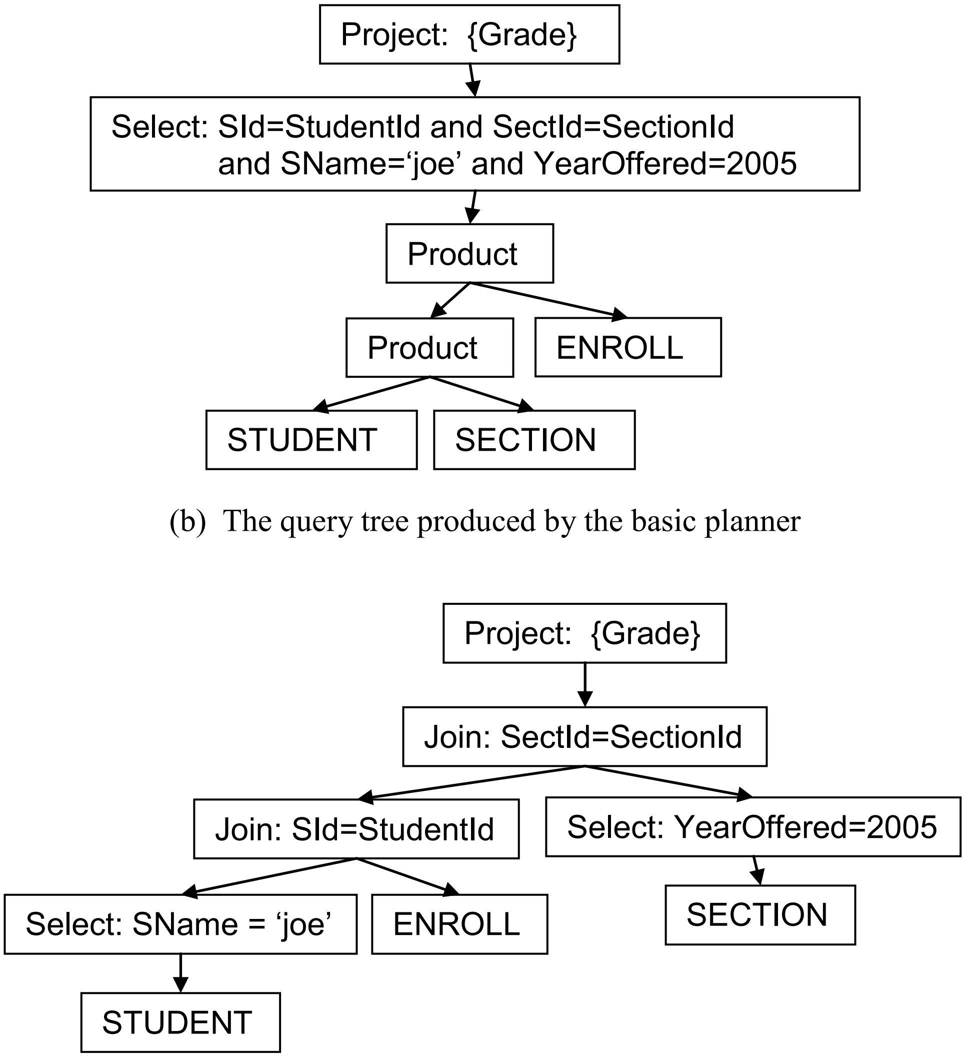
In general, an SQL query can have many equivalent query trees, and each node in the tree can be implemented in several ways. The planner has a lot of potential plans to choose from.

Clearly, it would be nice if the planner chose the most efficient plan (that is, the plan requiring the fewest block accesses). But doing so might entail a lot of work. Before we agree to do all of this work, we ought to be sure it is really worth the effort. What is so bad about using the basic planning algorithm of Chapter 19?

It turns out that different plans for the same query can require extremely different numbers of block accesses. Consider the example of Figure 24-11.

```
select Grade
from STUDENT, SECTION, ENROLL
where SId=StudentId and SectId=SectionId
and   SName='joe' and YearOffered=2005
```

(a) The SQL query



(b) The query tree produced by the basic planner

(c) An alternative query tree

Figure 24-11: Which query tree results in the better plan?

Part (a) of this figure is an SQL query that retrieves the grades that Joe received during 2005. Part (b) depicts the query tree created by the basic planner, and part (c) depicts an equivalent tree.

Consider first the plan from part (b). The basic query planner would assign a basic implementation to each node of the tree. Using the formulas from Figure 17-13, we can calculate the block accesses for this plan as follows. The first product (between STUDENT and SECTION) has 1,125,000,000 records and requires 112,504,500 block accesses. The second product therefore requires 56,250,112,504,500 block accesses. Since the *select* and *project* nodes require no additional block accesses, we see that this plan requires over 56 trillion block accesses. If we assume just 1 millisecond per block access, the database system would take about 1,780 years to answer this query.

Now consider the query tree from part (c), with the same pipelined plans for each node. Assume that there is one student named 'joe'. In this case, the selection on STUDENT requires 4,500 block accesses and outputs 1 record. The join with ENROLL requires 50,000 additional block accesses and outputs 34 records. And the join with SECTION requires 85,000 additional block accesses. The total number of block accesses is thus 139,500, which at 1 millisecond per access would take about 2.3 minutes.

The cost reduction from 1,780 years to 2.3 minutes is nothing short of amazing, and demonstrates how utterly worthless the basic planning algorithm is. No client can afford to wait a thousand years to get the answer to a query. If a database system is to be useful, its planner must be sophisticated enough to construct reasonable query trees.

Although 2.3 minutes is not an intolerable execution time, the planner can do even better by using other implementations for the nodes in the query tree. Consider again the query tree from part (c), and assume that ENROLL has an index on *StudentId*. The plan of Figure 24-12 is then possible.

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();

// the plan for the STUDENT node
Plan p1 = new TablePlan("student", tx);

// the plan for the select node above STUDENT
Predicate joepred = new Predicate(...); //sname='joe'
Plan p2 = new SelectPlan(p1, joepred);

// the plan for the ENROLL node
Plan p3 = new TablePlan("enroll", tx);

// an indexjoin plan between STUDENT and ENROLL
MetadataMgr mdMgr = SimpleDB.mdMgr();
Map<String,IndexInfo> indexes = mdMgr.getIndexInfo("enroll", tx);
IndexInfo ii = indexes.get("studentid");
Plan p4 = new IndexJoinPlan(p2, p3, ii, "sid", tx);

// the plan for the SECTION node
Plan p5 = new TablePlan("section", tx);

// the plan for the select node above SECTION
Predicate sectpred = new Predicate(...); //yearoffered=2005
Plan p6 = new SelectPlan(p5, sectpred, tx);

// a multibuffer product plan between the indexjoin and SECTION
Plan p7 = new MultiBufferProductPlan(p4, p6, tx);

// the plan for the select node above the multibuffer product
Predicate sectpred = new Predicate(...); //sectid=sectionid
Plan p8 = new SelectPlan(p7, sectpred, tx);

// the plan for the project node
List<String> fields = Arrays.asList("grade");
Plan p9 = new ProjectPlan(p8, fields, tx);
```

Figure 24-12: An efficient plan for the tree of Figure 24-11(c)

Most of the plans in this figure are basic plans. The two exceptions are p4 and p7. Plan p4 performs an index join; that is, for each selected STUDENT record, the index on *StudentId* is searched to find the matching ENROLL records. Plan p7 performs the join using a multibuffer product. It materializes its right side table (i.e. the sections from 2005), divides them into chunks, and performs the product of p4 with these chunks.

Let’s calculate the block accesses required by this plan. Plan p2 requires 4,500 block accesses and outputs 1 record. The index join accesses ENROLL once for each of the 34 records matching Joe’s STUDENT record; that is, the join requires 34 additional block accesses and outputs 34 records. Plan p6 (which finds the sections from 2005) requires 2,500 block accesses and outputs 500 records. The multibuffer product materializes these records, which requires 50 additional blocks to create a 50-block temporary table. Assuming that there are at least 50 buffers available, this temporary table fits into a single chunk, and so the product requires 50 more block accesses to scan the temporary table, in addition to the cost of computing the left-side records. The remaining plans require no additional block accesses. Thus the plan requires 7,134 total block accesses, which takes a little more than 7 seconds.

In other words, a careful choice of node implementations reduced the execution time of the query by a factor of almost 20, using the same query tree. This reduction may not be as dramatic as the difference we got when we used different query trees, but it is nevertheless substantial and important. Certainly, a commercial database system that is 20 times slower than its competition will not last long in the marketplace.

24.3 The Structure of a Query Optimizer

Given an SQL query, the planner must try to find the plan for that query that requires the fewest block accesses. This process is called *query optimization*.

> *Query optimization denotes the steps taken by the planner*
> *to find the most efficient plan for a query.*

But how can the planner determine that plan? An exhaustive enumeration of all possible plans is daunting: If a query has *n product* operations, then there are *(2n)!/n!* ways to arrange them, which means that the number of equivalent plans grows super-exponentially with the size of the query. And that's not even considering the different ways to place the nodes for the other operators, and the different ways to assign implementations to each node.

One way that the query planner deals with this complexity is to perform the optimization in two independent stages:

- Stage 1: Find the *most promising tree* for the query; that is, the query tree that seems most likely to produce the most efficient plan.
- Stage 2: Choose the best implementation for each node in that query tree.

By performing these stages independently, the planner reduces the choices it needs to make at each stage, which allows each stage to be simpler and more focused.

During each of these two optimization stages, the planner can reduce complexity even further by using *heuristics* to restrict the set of trees and plans that it considers.

> *A heuristic is a "rule of thumb" used for making decisions.*

For example, query planners typically use the heuristic "perform selections as early as possible". Experience has shown that in the optimal plan for a query, the *select* nodes are always (or nearly always) placed as early as possible. Thus by following this heuristic, a query planner does not need to consider any other placement of *select* nodes in the query trees it considers.

The following two sections examine the two stages of query optimization and their relevant heuristics.

24.4 Finding the Most Promising Query Tree

24.4.1 The cost of a tree

The first stage of query optimization is to find the "most promising" query tree.

> *The most promising query tree is the tree that the planner thinks will have the lowest-cost plan.*

The reason that the planner cannot actually determine the best tree is that cost information is not available during the first stage. Block accesses are associated with plans, and plans are not considered until the second stage.

Consequently, the planner needs a way to compare query trees without actually computing block accesses. The insight is to note that:

- nearly all of the block accesses in a query are due to *product* and *join* operations;
- the number of block accesses required by these operations is related to the size of their inputs[†].

The planner therefore defines the *cost* of a query tree as follows:

> *The cost of a query tree is the sum of the sizes of the inputs to each product/join node in the tree.*

For example, let's calculate the cost of the two query trees in Figure 24-11. These trees have two *product* nodes, so we sum the sizes of the inputs to each one. The results appear in Figure 24-13, and indicate that the second query tree is much better than the first one.

Query Tree	**Size of the inputs to the bottom *product* node**	**Size of the inputs to the top *product* node**	**Total cost of the tree**
Figure 24-11(b)	45,000 + 25,000	1,125,000,000 + 1,500,000	1,126,570,000
Figure 24-11(c)	1 + 1,500,000	34 + 25,000	1,525,035

Figure 24-13: Calculating the cost of two query trees

We can think of the cost of a query tree as a "quick and dirty" approximation of its value. The cost does not help us estimate block accesses, but it does help us determine the relative value of two trees. In particular, given two query trees, we expect that the most

[†] An exception is the index join, whose cost is basically unrelated to the size of the indexed table. The planner ignores that exception at this point.

efficient plan will come from the lower-cost tree. This expectation is not always correct (see Exercise 24.8). However, experience shows that it is correct most of the time, and even when it is not, the cheapest plan for the lower-cost tree tends to be good enough.

24.4.2 Pushing *select* and *groupby* down the tree

We now consider heuristics that the planner can use as it searches for the most promising query tree.

Our first heuristic concerns the placement of *select* nodes in the tree. The selection predicate comes from the *where* clause of an SQL query. Recall that the equivalences of Section 24.1.2 allow the planner to place *select* nodes in a query tree as follows:

- It rewrites the predicate into CNF.
- It creates a *select* node for each conjunct of the CNF predicate.
- It places the *select* nodes anywhere in the tree that it wants, provided that the predicate is meaningful at that point.

This third bullet point describes the choices available to the planner. Which placement of *select* nodes leads to the lowest-cost tree? We note that the output of a *select* node is less than (or equal to) the size of its input. So if we place a *select* node inside of a *product* or *join*, the inputs to those nodes will be smaller, and the cost of the tree will be reduced. This leads us to the following heuristic.

> Heuristic 1: *The planner only needs to consider query trees*
> *in which selections are pushed down as far as possible.*

Suppose that after pushing selections completely, two selections are next to each other in the query tree. Heuristic 1 does not specify the order these selections should appear in. However, the order makes no difference in the cost of the tree, and so the planner is free to choose any order, or to combine them into a single *select* node.

The placement of the *groupby* node in a query tree is similar. The planner of Chapter 19 placed the *groupby* node near the top of the tree, after the *product* and *select* nodes. However, the equivalence of Section 24.1.4 gives conditions under which the node can be pushed inside of the *product* nodes. As with selections, the output of a *groupby* node is usually smaller than its input, and is never larger than it. Consequently, placing the node as early as possible will lower the cost of the query tree.

> Heuristic 2: *The planner only needs to consider query trees*
> *in which aggregations are pushed down as far as possible.*

Heuristics 1 and 2 reduce the planner's task so that it doesn't have to worry about where to place *select* and *groupby* nodes. Given a query plan for the other operators, the placement of these nodes is well-specified.

24.4.3 Replacing *select-product* by *join*

Consider a join predicate involving fields from tables T_1 and T_2. When a *select* node containing this predicate is pushed down the tree, it will come to rest at a particular spot in the tree – namely the *product* node for which T_1 appears in one subtree, and T_2 appears in the other subtree. This pair of *select-product* nodes can be replaced by a single *join* node.

> Heuristic 3: *The planner should replace each select-product node pair in the query tree with a single join node.*

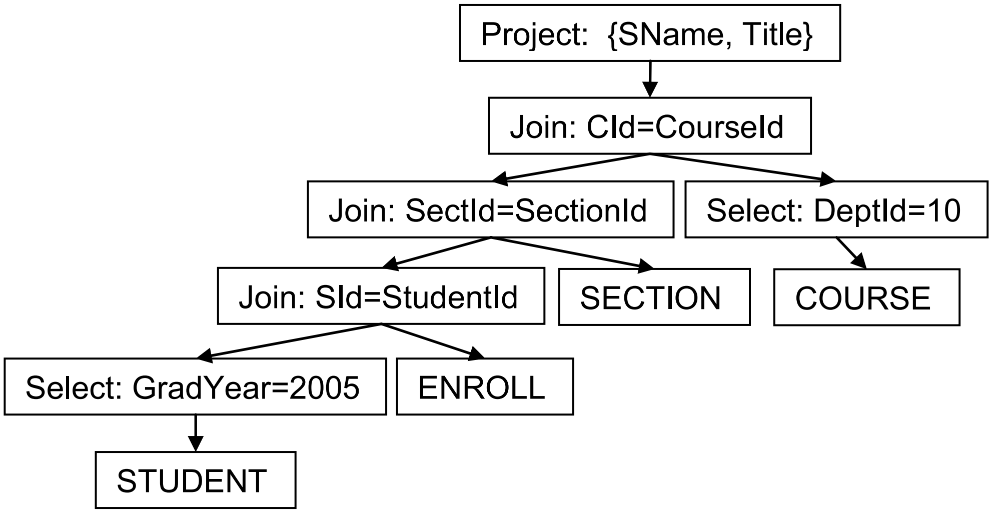
Although this heuristic does not change the cost of the query tree, it is an important step towards finding the best plan. As we have seen, there are several efficient implementations of the join operator. By identifying the joins in the query tree, the planner allows these implementations to be considered during the second stage of optimization.

24.4.4 Using left-deep query trees

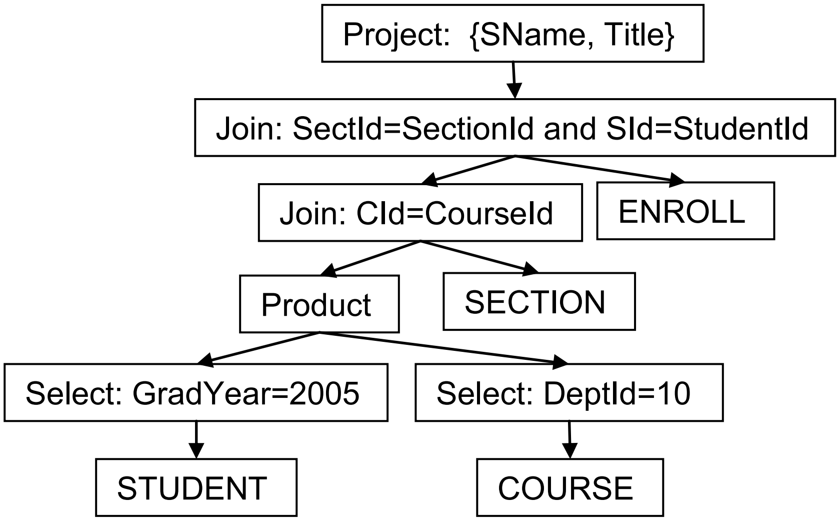
We now examine how the planner can choose the order in which the *product/join* operations should be performed. Consider the example of Figure 24-14. The SQL query of part (a) retrieves the name of the students graduating in 2005 and the titles of the math courses they took. Parts (b)-(f) depict five equivalent trees for this query.

```
select SName, Title
from STUDENT, ENROLL, SECTION, COURSE
where SId=StudentId and SectId=SectionId
and CId=CourseId and GradYear=2005 and DeptId=10
```

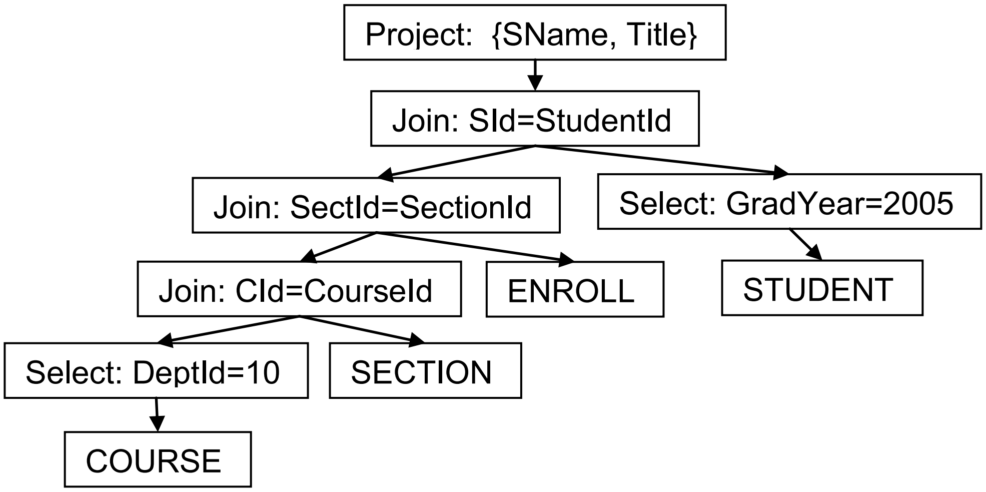
(a) The SQL query



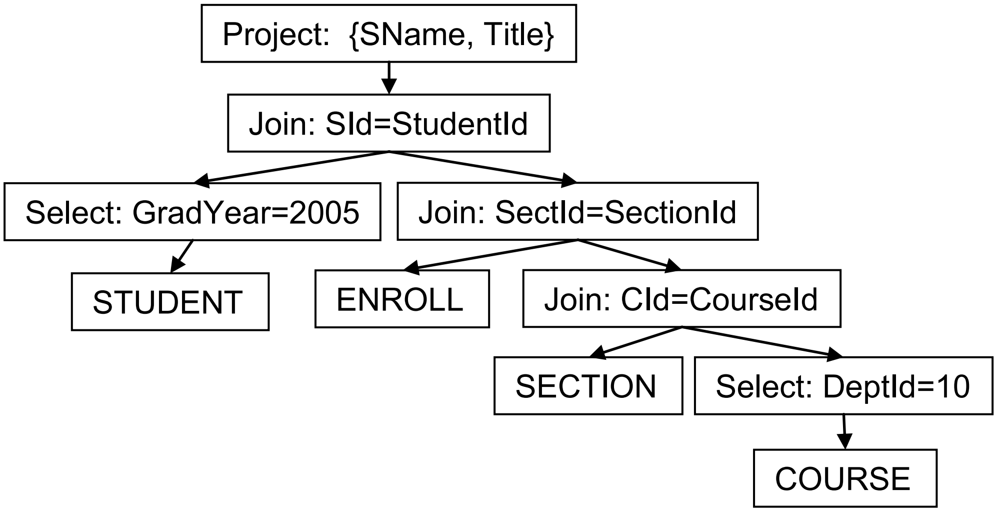
(b) A left-deep query tree



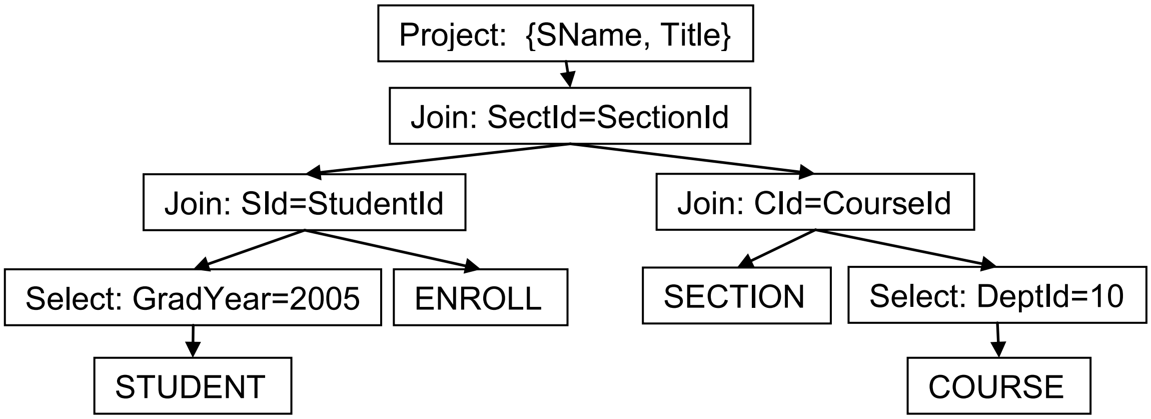
(c) An alternative left-deep query tree



(d) Yet another alternative left-deep query tree



(e) A right-deep query tree



(f) A bushy query tree

Figure 24-14: Equivalent query trees having different skeletons

Figure 24-15 computes the cost of each tree, using the statistics of Figure 16-7.

Tree	Cost of lower join	Cost of middle join	Cost of upper join	Total cost
(b)	1,500,900	55,000	30,013	1,585,913
(c)	913	36,700	3,750,000	3,787,613
(d)	25,013	1,500,625	38,400	1,564,038
(e)	25,013	1,500,625	38,400	1,564,038
(f)	1,500,900 (the left-hand join)	25,013 (the right-hand join)	30,625	1,556,538

Figure 24-15: The cost of the trees in Figure 24-14

These trees have different *skeletons*. The trees of parts (b)-(d) are called *left-deep*, because the right-side of each *product/join* node contains no other *product/join* nodes. Similarly, the tree of part (e) is called *right-deep*. The tree of part (f) is called *bushy*, because it is neither left-deep nor right-deep.

Many query planners adopt the following heuristic:

Heuristic 4: *The planner only needs to consider left-deep query trees.*

The reasoning behind this heuristic is not obvious. For example if we consider the trees of Figure 24-14, the lowest-cost tree is the bushy one. Moreover, that tree turns out to be the most promising one (see Exercise 24.9). So why would the planner deliberately choose to ignore a large set of trees that might contain the most promising one? There are two reasons.

The first reason is that left-deep trees tend to have the most efficient plans, even if they don't have the lowest cost. Think back to the join algorithms we have seen; they all work best when the right-side of the join is a stored table. For example, multibuffer product needs its right-side table to be materialized; so if the table is already stored, then additional materialization may not be necessary. And an index join is possible only when its right side is a stored table. Therefore, by using a left-deep tree, the planner increases the likelihood that it will be able to use more efficient implementations when it generates the final plan. Experience has shown that the best left-deep plan for a query tends to be either optimal or close enough to it.

The second reason is convenience. If a query has *n product/join* nodes, then there are only *n!* left-deep trees, which is far fewer than the *(2n)!/n!* possible trees. Heuristic 4 thus allows the planner to work much more quickly (which is important), with little risk of getting stuck with a bad plan.

A left-deep tree can be specified by listing its tables in order. The first table is the table that appears on the left-side of the bottommost *product/join* node, and the subsequent tables come from the right sides of each *product/join* node moving up the tree. This order is called the *join order* of the left-deep tree.

> *The join order of a left-deep query tree is a list of its tables.*
> - *The first table comes from the left side of the bottommost product/join node.*
> - *The subsequent tables come from the right sides of each product/join node, from the bottom of the tree to its top.*

For example, the left-deep tree of Figure 24-14(b) has the join order (STUDENT, ENROLL, SECTION, COURSE), and the tree of Figure 24-14(c) has the join order (STUDENT, COURSE, SECTION, ENROLL). Heuristic 4 therefore simplifies the job of the query planner – all the planner has to do is determine the best join order. Heuristics 1 to 4 then completely determine the corresponding query tree.

24.4.5 Choosing a join order heuristically

We now come to the most critical part of the query optimization process: finding the best join order for a given query. By "critical", we mean two things:

- The choice of join order dramatically effects the cost of the resulting query tree. An example is in Figure 24-14, where tree (b) is so much better than tree (c).
- There are so many possible join orders that it is usually not feasible to examine them all. In particular, a query that mentions *n* tables can have *n!* join orders.

Thus the planner must be very clever about which join orders it considers, so as not to get stuck with a bad one.

Two general approaches have been developed for determining good join orders: an approach that uses heuristics, and an approach that considers all possible orders. In this section we shall examine the heuristic approach; the next section considers exhaustive search.

The heuristic approach constructs the join order incrementally. That is, the planner begins by choosing one of the tables to be first in the join order. It then chooses another table to be next in the join order, and repeats until the join order is complete.

The following heuristic helps the planner to weed out the "obviously bad" join orders:

> Heuristic 5: *Each table chosen in the join order should join with the previously chosen tables, whenever possible.*

In other words, this heuristic states that the only *product* nodes in a query tree should correspond to joins. Note that the query tree of Figure 24-14(c) violates this heuristic, because of the product between the STUDENT and COURSE tables.

Why are join orders that violate Heuristic 5 so bad? Recall that the role of a join predicate is to filter out the meaningless output records generated by a *product* operation. So when a query tree contains a non-join *product* node, its intermediate tables will continue to propagate these meaningless records until the join predicate is encountered. For example, consider again the query tree of Figure 24-14(c). The product between STUDENT and COURSE results in 11,700 output records, because each of the 13 COURSE records from the math department is repeated 900 times (once for each student graduating in 2005). When this output table is joined with SECTION, each COURSE record is matched with its SECTION record; however, these matchings are repeated 900 times. Consequently, the output of that join is 900 times larger than it should be. When we add ENROLL to the join order, the join predicate with STUDENT finally kicks in, and the repetition is eliminated.

This example demonstrates that the output of a query tree involving a *product* node can start out small, but eventually the repetition caused by the product leads to a very high-cost tree. Thus Heuristic 5 asserts that *product* operations should be avoided if at all possible. Of course, if the user specifies a query that does not completely join all of the tables, then a *product* node will be inevitable. In this case, the heuristic ensures that this node will be as high in the tree as possible, so the repetition will have the smallest possible effect.

Heuristic 5 is a commonly-used heuristic. It is possible to find queries whose most promising query tree violates this heuristic (see Exercise 24.11), but such queries rarely occur in practice.

We now come to the important questions of which table to choose first, and which of the joinable tables to choose next. These are tough questions. The database community has proposed many heuristics, with very little consensus on which is most appropriate. We shall consider two logical possibilities, which we'll call Heuristics 6a and 6b:

> Heuristic 6a: *Choose the table that produces the smallest output.*

Heuristic 6a is the most direct, straightforward approach. The intention is this: Since the cost of a query tree is related to the sum of the sizes of its intermediate output tables, a good way to minimize this sum is to minimize each of those tables.

Let's use this heuristic on the query of Figure 24-14(a). The first table in the join order is COURSE, because its selection predicate reduces it to 13 records. The remaining tables are determined by Heuristic 5. That is, SECTION is the only table that joins with COURSE, and then ENROLL is the only table that joins with SECTION, which leaves STUDENT to be last in the join order. The resulting table appeared in Figure 24-14(d).

An alternative heuristic is the following:

> Heuristic 6b: *Choose the table that has the most restrictive selection predicate.*

Heuristic 6b arises from the insight that a selection predicate will have the greatest impact when it appears lowest in the query tree. For example, consider the query tree of Figure 24-14(b), and its selection predicate on STUDENT. That selection predicate has an obvious benefit: it reduces the number of STUDENT records, which lowers the cost of the *join* node immediately above it. But it has an even more important benefit – The predicate also reduces the output of that join from 1,500,000 records to just 30,000 records, which lowers the cost of each subsequent *join* node in the tree. In other words, the cost savings produced by a *select* node is compounded all the way up the tree. In contrast, the selection predicate on COURSE at the top of the tree has much less of an impact.

The tree of Figure 24-14(d) has the selection predicate on COURSE at the bottom of the tree, and the predicate on STUDENT at the top. The predicate on COURSE will therefore reduce the cost of every join in the tree, whereas the selection predicate on STUDENT will have less of an impact.

Since the selection predicates that are lower in a query tree have the greatest effect on its cost, it makes sense for the optimizer to choose the table whose predicate has the largest reduction factor. This is exactly what Heuristic 6b does. For example, the query tree of Figure 24-14(b) satisfies this heuristic. The first table in its join order is STUDENT, because its selection predicate reduces the table by a factor of 50, whereas the selection predicate for COURSE reduces it by only a factor of 40. The remaining tables in the join order, as before, are determined by Heuristic 5.

In this example, it turns out that using Heuristic 6b results in a lower-cost query tree than Heuristic 6a. This is typical. Studies (such as [Swami 1989]) have shown that although

Heuristic 6a makes intuitive sense and produces reasonable query trees, these trees tend to have higher cost than those from Heuristic 6b.

24.4.6 Choosing a join order by exhaustive enumeration

Heuristics 5 and 6 tend to produce good join orders, but are not guaranteed to produce the best one. If a vendor wants to be sure that its planner finds the optimum join order, its only alternative is to enumerate all of them. This section considers such a strategy.

A query that mentions *n* tables can have as many as *n!* join orders. A well-known algorithmic technique, known as *dynamic programming*, can reduce the time needed to find the most promising join order to O(2^n). If *n* is reasonably small (say, not more than 15 or 20 tables), then this algorithm is efficient enough to be practical.

For an illustration of how this technique can save time, consider a query that joins all five tables in our example database. Four of its 120 possible join orders are:

```
(STUDENT, ENROLL, SECTION, COURSE, DEPT)
(STUDENT, SECTION, ENROLL, COURSE, DEPT)
(STUDENT, ENROLL, SECTION, DEPT, COURSE)
(STUDENT, SECTION, ENROLL, DEPT, COURSE)
```

The first two join orders differ only on their second and third tables. Suppose we determine that the partial join order (STUDENT, ENROLL, SECTION) has a lower cost than (STUDENT, SECTION, ENROLL). Then it follows, without any further calculation, that the first join order must have lower cost than the second one. Moreover, we also know that the third join order requires fewer block accesses than the fourth one. And in general, we know that any join order that begins (STUDENT, SECTION, ENROLL) is not worth considering.

The dynamic programming algorithm uses an array variable named *lowest*, which has an entry for each possible set of tables. If *S* is a set of tables, then *lowest[S]* contains three values:

- the lowest-cost join order involving the tables in *S*;
- the cost of the query tree corresponding to that join order;
- the number of records output by that query tree.

The algorithm begins by computing *lowest[S]* for each set of two tables, then each set of three tables, and continues until it reaches the set of all tables. The optimum join order is the value of *lowest[S]* when S is the set of all tables.

Computing sets of 2 tables

Consider a set of 2 tables, say {T_1, T_2}. The value of *lowest[{T_1, T_2}]* is determined by computing the cost of the query tree that takes the join (or product, if there is no join predicate) of the two tables and their selection predicates. The cost of the query tree is the sum of the sizes of the two inputs to the *product/join* node. Note that the cost is the same regardless of which table is first. Thus, the planner must use some other criterion to determine the first table. A reasonable choice is to use Heuristic 6a or 6b.

Computing sets of 3 tables

Consider a set of 3 tables, say {T_1, T_2, T_3}. Their lowest-cost join order can be computed by considering the following join orders:

```
lowest[{$T_2$, $T_3$}] joined with $T_1$
lowest[{$T_1$, $T_3$}] joined with $T_2$
lowest[{$T_1$, $T_2$}] joined with $T_3$
```

The join order having the lowest cost will be saved as the value of *lowest[{T_1, T_2, T_3}]*.

Computing sets of n tables

Now suppose that the variable *lowest* has been calculated for each set of n-1 tables. Given the set {T1, T2, …, Tn}, the algorithm considers the following join orders:

```
lowest[{$T_2$, $T_3$ ,…, $T_n$}]  joined with $T_1$
lowest[{$T_1$, $T_3$ ,…, $T_n$}]  joined with $T_2$
. . .
lowest[{$T_1$, $T_2$ ,…, $T_{n-1}$}] joined with $T_n$
```

The join order having the lowest cost is the best join order for the query.

As an example, let’s use the dynamic programming algorithm on the query of Figure 24-14. The algorithm begins by considering all 6 sets of two tables, as shown in Figure 24-16(a).

S	Partial Join Order	Cost	#Records
{ENROLL,STUDENT}	(STUDENT,ENROLL)	1,500,900	30,000
	(ENROLL,STUDENT)	1,500,900	
{ENROLL,SECTION}	(SECTION,ENROLL)	1,525,000	1,500,000
	(ENROLL,SECTION)	1,525,000	
{COURSE,SECTION}	(COURSE,SECTION)	25,500	25,000
	(SECTION,COURSE)	25,500	
{SECTION,STUDENT}	(STUDENT,SECTION)	25,900	22,500,000
	(SECTION,STUDENT)	25,900	
{COURSE,STUDENT}	(STUDENT,COURSE)	1,400	450,000
	(COURSE,STUDENT)	1,400	
{COURSE,ENROLL}	(COURSE,ENROLL)	1,500,500	450,000,000
	(ENROLL,COURSE)	1,500,500	

(a) All sets of two tables

S	Partial Join Order	Cost	#Records
{ENROLL,SECTION,STUDENT}	(STUDENT,ENROLL,SECTION)	1,555,900	30,000
	(SECTION,ENROLL,STUDENT)	3,025,900	
	(STUDENT,SECTION,ENROLL)	24,025,900	
{COURSE,ENROLL,STUDENT}	(STUDENT,ENROLL,COURSE)	1,531,400	15,000,000
	(STUDENT,COURSE,ENROLL)	1,951,400	
	(COURSE,ENROLL,STUDENT)	451,501,400	
{COURSE,ENROLL,SECTION}	(SECTION,ENROLL,COURSE)	1,500,500	1,500,000
	(COURSE,SECTION,ENROLL)	1,550,500	
	(COURSE,ENROLL,SECTION)	450,025,000	
{COURSE,SECTION,STUDENT}	(COURSE,SECTION,STUDENT)	25,900	22,500,000
	(STUDENT,COURSE,SECTION)	475,000	
	(STUDENT,SECTION,COURSE)	22,500,500	

(b) All sets of three tables

Join Order	Cost
(STUDENT,ENROLL,SECTION,COURSE)	1,586,400
(COURSE,SECTION,ENROLL,STUDENT)	3,051,400
(STUDENT,ENROLL,COURSE,SECTION)	16,556,400
(COURSE,SECTION,STUDENT,ENROLL)	24,051,400

(c) All sets of four tables

Figure 24-16: Calculating the best join order for Figure 24-14

Each set of two tables has two partial join orders, which are listed in the row corresponding to that set. The join orders for each set are listed in terms of desirability.

In this case, they have the same cost, so they are listed according to Heuristic 6b. The first partial join order for each set is chosen as the representative of that set.

The algorithm then considers all four sets of three tables. Figure 24-16(b) lists the partial join orders for these sets, and their costs. Each set has three possible join orders. The first two tables in the join order are the lowest-cost representative of their set from Figure 29-16(a). The costs are listed from lowest to highest cost, so the first partial join order for each set is chosen as the representative of that set.

Figure 24-16(c) considers sets of four tables. There are four join orders to consider. The first three tables in each join order represent the lowest-cost join order from Figure 24-16(b); the fourth table in the join order is the missing table. This table shows that the join order (STUDENT, ENROLL, SECTION, COURSE) is optimum.

Note that at each stage, the algorithm must compute the value of *lowest* for every possible set of prefix tables, because there is no way of knowing how the costs will change during subsequent stages. It may be that the prefix that has highest cost at one stage will produce the lowest-cost join order overall, because of how the remaining tables join with it.

24.5 Finding the Most Efficient Plan

The first stage of query optimization was to find the most promising query tree. We now consider the second stage, which is to turn that query tree into an efficient plan.

The planner constructs the plan by choosing an implementation for each node in the query tree. It chooses these implementations bottom-up, starting from the leaves. The advantage of proceeding bottom-up is that when a given node is considered, the planner will have already chosen the lowest-cost plan for each of its subtrees. The planner can thus consider each possible implementation of the node, use the implementation's *blocksAccessed* method to calculate the cost of that implementation, and choose the implementation having the lowest cost.

Note that the planner chooses the implementation of each node independently of the implementations of the other nodes. In particular, it does not care how the subtrees of a node are implemented; it only needs to know the cost of that implementation. This lack of interaction between nodes significantly reduces the computational complexity of plan generation. If the query tree has *n* nodes, and each node has at most *k* implementations, then the planner needs to examine at most *k*n* plans, which is certainly reasonable.

Nevertheless, the planner can also take advantage of heuristics to speed up plan generation. These heuristics tend to be operation-specific. For example:

Heuristic 7: *If possible, use indexselect to implement a select node.*

Heuristic 8: *Implement a join node according to the following priority:*
- *Use indexjoin if possible.*
- *Use hashjoin if one of the input tables is small.*
- *Use mergejoin otherwise.*

We have one more issue to consider. Whenever the planner chooses to implement a node using a materialized plan, then it should also insert *project* nodes into the query tree, as follows:

Heuristic 9: *The planner should add a project node as the child of each materialized node, in order to remove fields that are no longer needed.*

Heuristic 9 ensures that the temporary tables created by a materialized implementation are as small as possible. There are two reasons why this is important: a larger table takes more block accesses to create, and a larger table also takes more block accesses to scan. The planner therefore should determine which fields will be needed by the materialized node and it ancestors, and insert a *project* node to remove the other fields from its input.

For example, consider the query tree of Figure 24-17. This tree returns the grades that Joe received in 2005, and is equivalent to the trees of Figure 24-11.

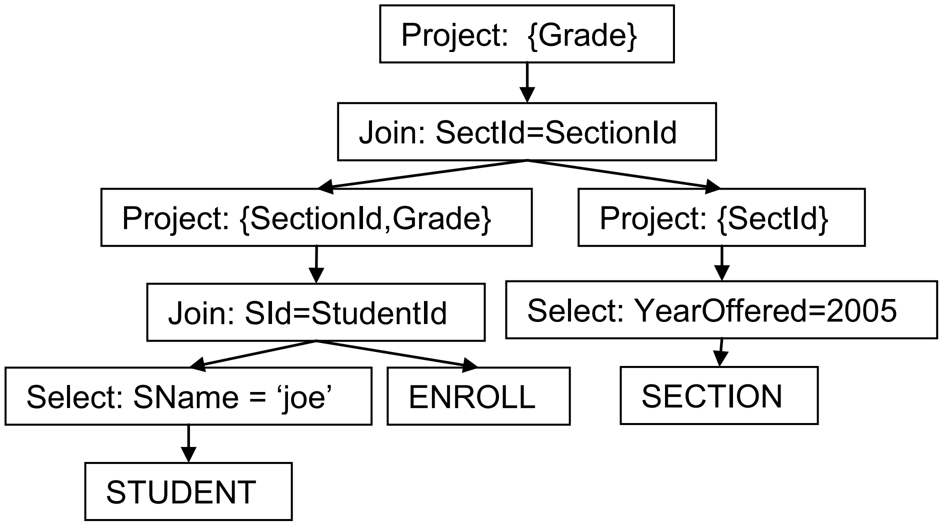


Figure 24-17: A query tree for the query of Figure 24-11, with added *project* nodes

When we created the plan of Figure 24-12, we chose to implement the upper *join* node with multibuffer product, which is materialized. Heuristic 9 asserts that *project* nodes need to be added to the query tree as children of that *join* node; these nodes are shown in Figure 24-17. The right-hand *project* node is especially important, because it reduces the size of the temporary table by about 75%, thereby allowing the algorithm to run using fewer chunks.

24.6 Combining the Two Stages of Optimization

The easiest way to understand query optimization is as two separate stages: a first stage that constructs the query tree from the SQL query; and a second stage that constructs the plan from the query tree. In practice, however, these stages are often combined. There are two good reasons in favor of combining optimization stages:

- *convenience*: The plan can be created directly, without having to create an explicit query tree.
- *accuracy*: Since the plans are created concurrently with the query tree, it may be possible to calculate the cost of the tree in terms of actual block accesses.

This section examines two examples of combined optimization: the heuristic-based SimpleDB optimizer, and the enumeration-based "Selinger-style" optimizer.

24.6.1 The heuristic-based SimpleDB optimizer

The SimpleDB query optimizer is implemented in package *simpledb.opt* via the two classes *HeuristicQueryPlanner* and *TablePlanner*. In order to use this optimizer in SimpleDB, you must modify the method *SimpleDB.planner* in package *simpledb.server* so that it creates an instance of *HeuristicQueryPlanner* instead of *BasicQueryPlanner*.

The class *HeuristicQueryPlanner*

The class *HeuristicQueryPlanner* uses Heuristic 6a to determine the join order. Every table has a *TablePlanner* object. When a table is added to the join order, its *TablePlanner* object creates the corresponding plan, adding appropriate selection and join predicates, and using indexes when possible. In this way, the plan is built simultaneously with the join order.

The code for *HeuristicQueryPlanner* appears in Figure 24-18. The collection *tableinfo* contains a *TablePlanner* object for each table in the query. The planner begins by choosing (and removing) the object from this collection corresponding to the smallest table, and uses its select plan as the *current plan*. It then repeatedly chooses (and removes) from the collection the table having the lowest-cost join. The planner sends the current plan to that table's *TablePlanner* object, which creates and returns the join plan. This join plan then becomes the current plan. This process is continued until the collection is empty, at which point the current plan is the final one.

```
public class HeuristicQueryPlanner implements QueryPlanner {
   private Collection<TablePlanner> tableinfo =
                                       new ArrayList<TablePlanner>();

   public Plan createPlan(QueryData data, Transaction tx) {

   // Step 1:  Create a TablePlanner object for each mentioned table
      for (String tblname : data.tables()) {
         TablePlanner tp = new TablePlanner(tblname, data.pred(), tx);
         tableinfo.add(tp);
      }

   // Step 2:  Choose the lowest-size plan to begin the join order
      Plan currentplan = getLowestSelectPlan();

   // Step 3:  Repeatedly add a plan to the join order
      while (!tableinfo.isEmpty()) {
         Plan p = getLowestJoinPlan(currentplan);
         if (p != null)
            currentplan = p;
         else  // no applicable join
            currentplan = getLowestProductPlan(currentplan);
      }

   // Step 4.  Project on the field names and return
      return new ProjectPlan(currentplan, data.fields());
   }

   private Plan getLowestSelectPlan() {
      TablePlanner besttp = null;
      Plan bestplan = null;
      for (TablePlanner tp : tableinfo) {
         Plan plan = tp.makeSelectPlan();
         if (bestplan == null ||
             plan.recordsOutput() < bestplan.recordsOutput()) {
            besttp = tp;
            bestplan = plan;
         }
      }
      tableinfo.remove(besttp);
      return bestplan;
   }

   private Plan getLowestJoinPlan(Plan current) {
      TablePlanner besttp = null;
      Plan bestplan = null;
      for (TablePlanner tp : tableinfo) {
         Plan plan = tp.makeJoinPlan(current);
         if (plan != null && (bestplan == null ||
             plan.recordsOutput() < bestplan.recordsOutput())) {
            besttp = tp;
            bestplan = plan;
         }
      }
      if (bestplan != null)
         tableinfo.remove(besttp);
      return bestplan;
   }

   private Plan getLowestProductPlan(Plan current) {
      TablePlanner besttp = null;
      Plan bestplan = null;
      for (TablePlanner tp : tableinfo) {
         Plan plan = tp.makeProductPlan(current);
         if (bestplan == null ||
             plan.recordsOutput() < bestplan.recordsOutput()) {
            besttp = tp;
            bestplan = plan;
         }
      }
      tableinfo.remove(besttp);
      return bestplan;
   }
}
```

Figure 24-18: The code for the SimpleDB class *HeuristicQueryPlanner*

The class *TablePlanner*

An object of class *TablePlanner* is responsible for creating plans for a single table; its code appears in Figure 24-19. The *TablePlanner* constructor creates a table for the specified table and obtains the information about the indexes for the table, and saves the query predicate. The class has public methods *makeSelectPlan*, *makeProductPlan* and *makeJoinPlan*.

```
class TablePlanner {
   private TablePlan myplan;
   private Predicate mypred;
   private Schema myschema;
   private  Map<String,IndexInfo> indexes;
   private Transaction tx;

   public TablePlanner(String tblname, Predicate mypred,
                                       Transaction tx) {
      this.mypred  = mypred;
      this.tx  = tx;
      myplan   = new TablePlan(tblname, tx);
      myschema = myplan.schema();
      indexes  = SimpleDB.mdMgr().getIndexInfo(tblname, tx);
   }

   public Plan makeSelectPlan() {
      Plan p = makeIndexSelect();
      if (p == null)
         p = myplan;
      return addSelectPred(p);
   }

   public Plan makeJoinPlan(Plan current) {
      Schema currsch = current.schema();
      Predicate joinpred = mypred.joinPred(myschema, currsch);
      if (joinpred == null)
         return null;
      Plan p = makeIndexJoin(current, currsch);
      if (p == null)
         p = makeProductJoin(current, currsch);
      return p;
   }

   public Plan makeProductPlan(Plan current) {
      Plan p = addSelectPred(myplan);
      p = new MaterializePlan(p, tx);
      return new MultiBufferProductPlan(current, p, tx);
   }

   private Plan makeIndexSelect() {
      for (String fldname : indexes.keySet()) {
         Constant val = mypred.equatesWithConstant(fldname);
         if (val != null) {
            IndexInfo ii = indexes.get(fldname);
            return new IndexSelectPlan(myplan, ii, val, tx);
         }
      }
      return null;
   }

   private Plan makeIndexJoin(Plan current, Schema currsch) {
      for (String fldname : indexes.keySet()) {
         String outerfield = mypred.equatesWithField(fldname);
         if (outerfield != null && currsch.hasField(outerfield)) {
            IndexInfo ii = indexes.get(fldname);
            Plan p = new IndexJoinPlan(current, myplan, ii,
                                       outerfield, tx);
            p = addSelectPred(p);
            return addJoinPred(p, currsch);
         }
      }
      return null;
   }

   private Plan makeProductJoin(Plan current, Schema currsch) {
      Plan p = makeProductPlan(current);
      return addJoinPred(p, currsch);
   }

   private Plan addSelectPred(Plan p) {
      Predicate selectpred = mypred.selectPred(myschema);
      if (selectpred != null)
         return new SelectPlan(p, selectpred);
      else
         return p;
   }

   private Plan addJoinPred(Plan p, Schema currsch) {
      Predicate joinpred = mypred.joinPred(currsch, myschema);
      if (joinpred != null)
         return new SelectPlan(p, joinpred);
      else
         return p;
   }
}
```

Figure 24-19: The code for the SimpleDB class *TablePlanner*

The method *makeSelectPlan* creates a select plan for its table. The method first calls the private method *makeIndexSelect* to determine if an index can be used; if so, an *IndexSelect* plan is created. The method then calls the private method *addSelectPred*, which adds a select plan for the portion of the predicate that applies to the table.

Method *makeProductPlan* adds a select plan to the table plan, and then creates a *MultiBufferProductPlan* to implement the product of the specified plan with this plan[†].

Finally, *makeJoinPlan* is called. It first calls the predicate's *joinPred* method to determine if a join exists between the specified plan and this plan. If no join predicate exists, the method returns null. If a join predicate does exist, the method looks to see if an *IndexJoinScan* can be created. If not, then the join is implemented by creating a multibuffer product followed by a select.

Records output vs. blocks accessed

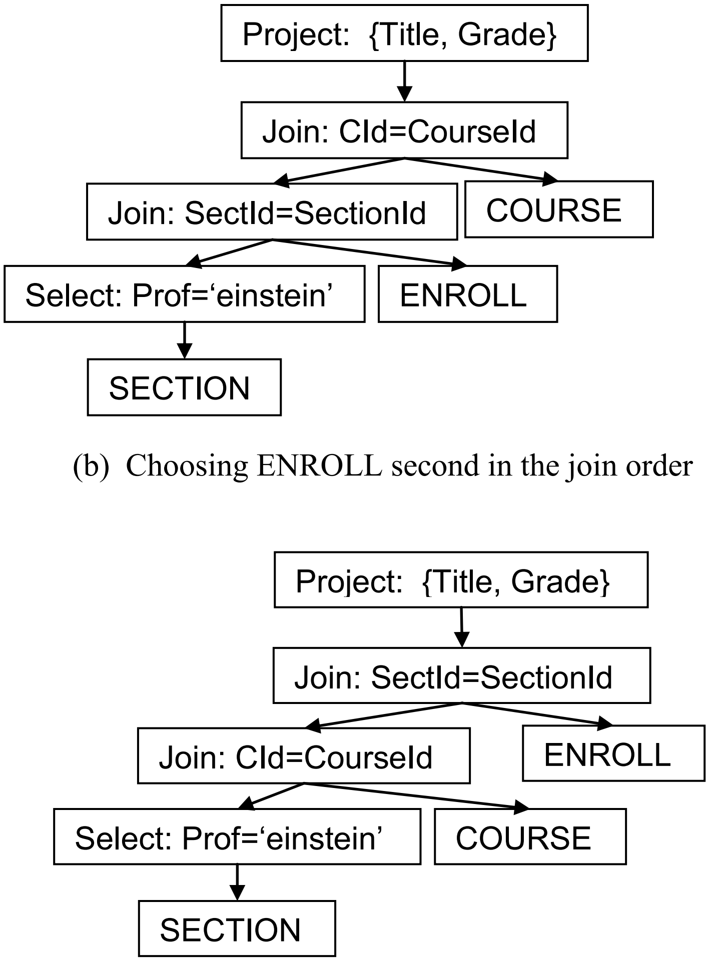
[†] Ideally, the method should create a hashjoin plan, but SimpleDB does not support hash joins. See Exercise 24.17.

The *HeuristicQueryPlanner* code calculates the lowest-cost plan using the method *recordsOutput*, even though it could have used the method *blocksAccessed*. That is, it attempts to find the plan needing the smallest number of block accesses without ever examining the block requirements of its subplans. This situation deserves explanation.

As we have seen, the problem with using heuristic optimization is that partial join orders that start out cheap can wind up very expensive, and the best join order may have a very expensive beginning. It is therefore important for the optimizer to not get sidetracked by a join that seems better than it is. Figure 24-20 illustrates this problem.

```
select Title, Grade
from ENROLL, SECTION, COURSE
where SectId=SectionId and CId=CourseId
and   Prof='einstein'
```

(a) The SQL query



(b) Choosing ENROLL second in the join order

(c) Choosing COURSE second in the join order

Figure 24-20: Which table should be second in the join order?

The query of Figure 24-20(a) returns the grades given and title for each course taught by professor Einstein. Assume the statistics of Figure 16-7, and suppose that ENROLL has an index on *SectionId*. The SimpleDB optimizer will choose SECTION to be first in the join order because it is smallest (as well as most selective). The question is which table it should choose next. If our criterion is to minimize records output, then we should choose COURSE. But if our criterion is to minimize blocks accessed, then we should choose ENROLL, because the index join will be more efficient. However, ENROLL turns out to be the wrong choice, because the high number of output records causes the subsequent join with COURSE to be much more expensive.

This example demonstrates that the large number of matching ENROLL records has a significant affect on the cost of subsequent joins; thus ENROLL should appear as late as possible in the join order. By minimizing records output, the optimizer ensures that ENROLL winds up at the end. The fact that the join with ENROLL has a fast implementation is misleading, and irrelevant.

24.6.2 Selinger-style optimization

The SimpleDB optimizer uses heuristics for choosing the join order. In the early 1970's, researchers at IBM wrote an influential optimizer for the System-R prototype database system; this optimizer chose its join orders using dynamic programming. That optimization strategy is often called "Selinger-style", in reference to Pat Selinger, who headed the optimizer team.

Selinger-style optimization combines dynamic programming with plan generation. In particular, the algorithm calculates *lowest[S]* for each set of tables S. But instead of saving a join order in *lowest[S]*, the algorithm saves the lowest-cost plan.

The algorithm begins by calculating the lowest-cost plan for each pair of tables. It then uses these plans to calculate the lowest-cost plan for each set of three tables, and so on, until it has calculated the overall lowest-cost plan.

In this algorithm, the lowest-cost plan is the plan having the fewest block accesses, and not the plan having the fewest output records. That means that this is the only algorithm we have seen that actually considers block accesses when choosing its join order; therefore, its estimates are likely to be more accurate than the other algorithms.

Why is Selinger-style optimization able to use block accesses? The reason is that unlike heuristic optimization, it considers all left-deep trees, and does not throw out a partial join order until it is sure that the order is not useful. Look back at the example of Figure 24-20. The Selinger-style algorithm will calculate and store the lowest plans for both {SECTION, ENROLL} and {SECTION, COURSE}, even though the plan for {SECTION, ENROLL} is cheaper. It considers both of those plans when it calculates the lowest plan for {ENROLL, SECTION, COURSE}. When it discovers that joining

COURSE to (ENROLL,SECTION) is excessively costly, it is able to use an alternative plan.

Another advantage to using block accesses to compare plans is that a more detailed cost analysis is possible. For example, the optimizer can take the cost of sorting into account. Consider the query tree of Figure 24-21.

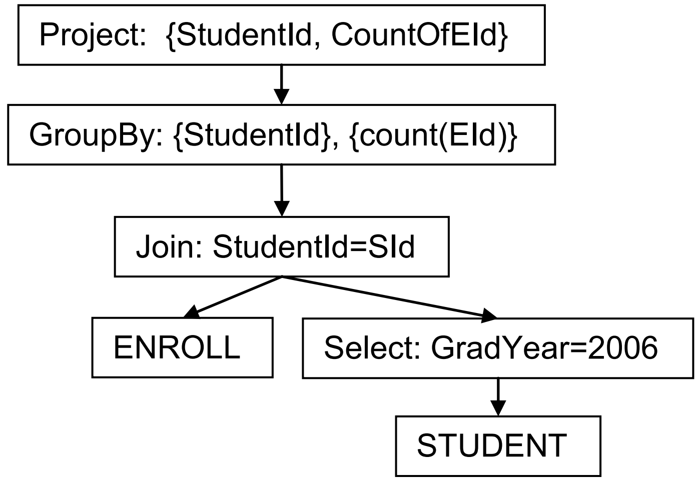

Project: {StudentId, CountOfEId}
GroupBy: {StudentId}, {count(EId)}
Join: StudentId=SId
ENROLL
Select: GradYear=2006
STUDENT

Figure 24-21: What is the best way to join ENROLL with STUDENT?

Suppose that we join ENROLL with STUDENT using a hashjoin. When we go to do the grouping, we will need to materialize the output and sort it on *StudentId*. Alternatively, suppose instead that we use a mergejoin to join the tables. In this case, we would not need to preprocess the output, because it would already be sorted on *StudentId*. In other words, it is possible that using a mergejoin could result in the best final plan, even if it was less efficient than the hashjoin!

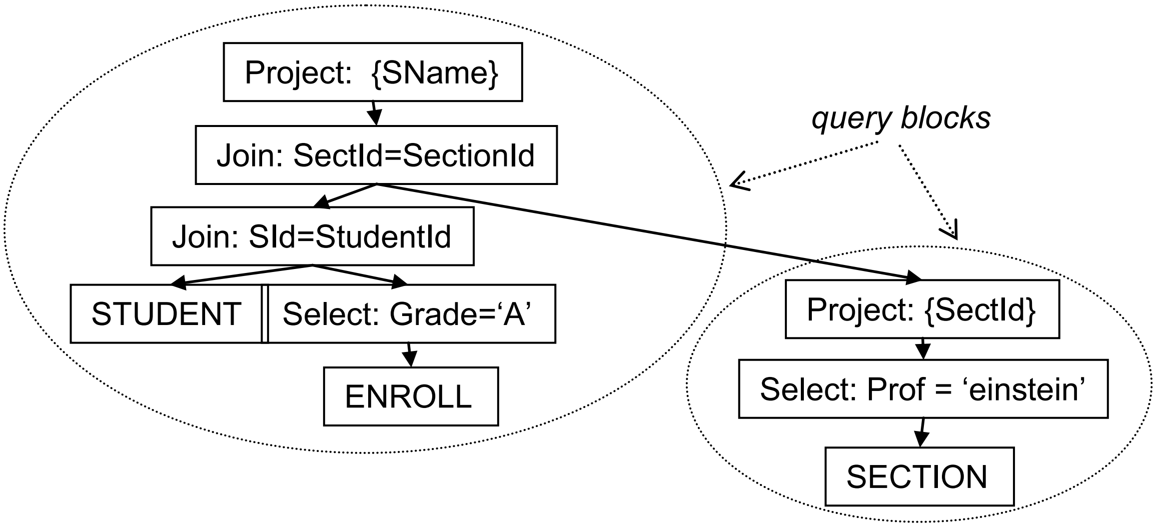
The point of this example is that we also need to keep track of sort order if we want to generate the best plan. A Selinger-style optimizer can do so by saving the lowest-cost plan for each sort order in *lowest[S]*. In the above example, the value of *lowest[{ENROLL,STUDENT}]* will contain both the mergejoin and the hashjoin plans, because each has a different sort order.

24.7 Merging query blocks

We now consider how to optimize queries that mention views. Consider for example the query of Figure 24-22(a), which uses a view to retrieve the names of the students who received an 'A' in a course taught by Professor Einstein. In Chapter 19 we saw how that the basic query planner creates the plan for such a query – It plans the view definition and the query separately, and then hooks the plan for the view into the plan for the query. That plan appears in Figure 24-22(b).

```
create view EINSTEIN as
select SectId
from SECTION
where Prof = 'einstein'

select SName
from STUDENT, ENROLL, EINSTEIN
where SId = StudentId and SectionId = SectId
and Grade = 'A'
```

(a) A view definition and a query that uses it



(b) Planning each query block separately

Figure 24-22: Planning a view query

The plan associated with each query and view definition is called a *query block*. The plan for Figure 24-22(b) illustrates the simplest way that an optimizer can deal with view queries: It can optimize each query block separately before combining them into the final plan. Although separate optimization is simple to implement, the plans that get created are not necessarily very good. The plan of Figure 24-22 is a case in point. The best join order is (SECTION, ENROLL, STUDENT), but this join order is not possible given these query blocks.

A solution to this problem is to *merge* the query blocks, and plan their contents as a single query. For example in Figure 24-22, the planner can ignore the *project* node of the view definition block, and add its *select* and table nodes to the main query. Such a strategy is possible if the view definition is sufficiently simple. The situation becomes much more complex when the view definition contains grouping or duplicate removal, and merging may not be possible.

24.8 Chapter Summary

- Two queries are *equivalent* if their output tables contain exactly the same records (although not necessarily in the same order), regardless of the contents of the database.

- An SQL query may have many equivalent query trees. These equivalences are inferred from properties of the relational algebra operators.
    - The *product* operator is commutative and associative. These properties imply that the *product* nodes in a query tree can be computed in any order.
    - A *select* node for predicate p can be split into several *select* nodes, one for each conjunct of p. Writing p in conjunctive normal form (CNF), allows it to be split into the smallest pieces. The nodes for each conjunct can be placed anywhere within the query tree, as long as their selection predicate is meaningful.
    - A pair of *select-product* nodes can be replaced by a single *join* node.
    - A *groupby* node can be moved inside of a relationship join, provided that the groups can be determined by the foreign key field and the aggregation functions are only calculated for fields of the foreign-key table.
    - A *project* node can be inserted over any node in a query tree, provided that its projection list contains all fields mentioned in the ancestors of the node.

- Plans for two equivalent trees can have radically different execution times. Therefore, a planner tries to find the plan that requires the fewest block accesses. This process is called *query optimization*.

- Query optimization is difficult because there can be far more plans for an SQL query than the planner can feasibly enumerate. The planner can deal with this complexity by performing the optimization in two independent stages:
    - Stage 1: Find the *most promising* tree for the query; that is, the query tree that seems most likely to produce the most efficient plan.
    - Stage 2: Choose the best plan for that query tree.

- During stage 1, the planner cannot estimate block accesses, because it does not know what plans are being used. Instead, it defines the *cost* of a query tree to be the sum of the sizes of the inputs to each product/join node in the tree. Intuitively, a low-cost query tree minimizes the size of intermediate joins. The idea is that the output of each join will be the input to the subsequent join; and so the larger the intermediate outputs, the more expensive it will be to execute the query.

- The planner also adopts *heuristics* to limit the set of trees and plans that it considers. Common heuristics are:
    - Place *select* and *groupby* nodes as deep as possible in the query tree.
    - Replace each *select-product* node pair by a *join* node.
    - Place a *project* node above the inputs to each materialized plan.
    - Consider only left-deep trees.

  - Whenever possible, avoid product operations that are not joins.

- Each left-deep tree has an associated *join order*. Finding a good join order is the most difficult part of query optimization.

- One way to choose a join order is to use heuristics. Two reasonable (but conflicting) heuristics are:
  - Choose the table that produces the smallest output.
  - Choose the table that has the most restrictive predicate.

  This second heuristic strives to create a query tree in which the most restrictive *select* nodes are maximally deep, the intuition being that such trees tend to have the lowest cost.

- Another way to choose a join order is to exhaustively examine all possible join orders, using dynamic programming. The dynamic programming algorithm calculates the lowest join order for each set of tables, starting with sets of 2 tables, then sets of 3 tables, and continues until it reaches the set of all tables.

- During the second optimization stage, the planner constructs the plan by choosing an implementation for each node in the query tree. It chooses each implementation independently of the implementations of the other nodes, and calculates its cost in terms of blocks accessed. The planner can determine the lowest-cost plan either by examining all possible implementations of each node, or by following heuristics such as:
  - Use indexing whenever possible.
  - If indexing is not possible for a join, then use a hashjoin if one of the input tables is small; otherwise use a mergejoin.

- An implementation of a query optimizer can combine its two stages, constructing the plan in conjunction with the query tree. The SimpleDB optimizer uses heuristics to determine the join order, and incrementally constructs the plan as each table is chosen. A *Selinger-style* optimizer uses dynamic programming – instead of saving the lowest cost join order for each set of tables, it saves the lowest-cost plan. The advantage of a Selinger-style optimizer is that, unlike any of the other techniques, it can use estimated block accesses to calculate the best join order.

- A query that uses a view will have a plan consisting of multiple *query blocks*. The most straightforward way to handle multiple query blocks is to optimize each one separately, and then combine them. However, more efficient plans are possible if the query blocks can be optimized together. Such as strategy is possible if the view definition is sufficiently simple.

24.9 Suggested Reading

This chapter gives a basic introduction to query optimization; the articles [Graefe 1993] and [Chaudhuri 1998] go into considerably more detail. The paper [Swami 1989] contains an experimental comparison of various join-order heuristics. The System-R optimizer is described in [Selinger et al 1979].

One difficulty with traditional query planners is that their heuristics and optimization strategy are hard-coded into their methods. Consequently, the only way to change the heuristics or to add new relational operators is to rewrite the code. An alternative approach is to express operators and their transformations as *rewrite rules*, and to have the planner repeatedly use the rules to transform the initial query into an optimum one. To change the planner, one then only needs to change the rule set. A description of this strategy appears in [Pirahesh 1992].

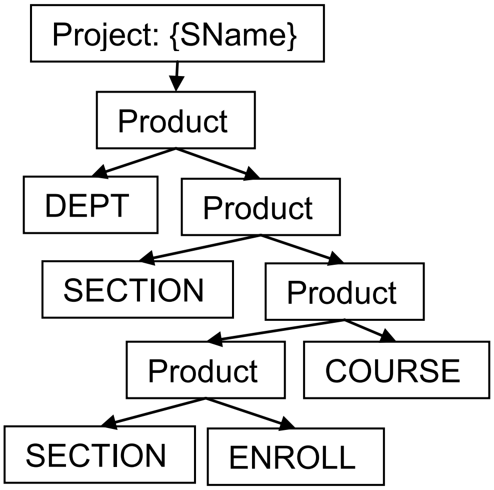
In the optimization strategies we have covered, there is a sharp distinction between query planning and query execution – once a plan is opened and executed, there is no turning back. If the planner has mistakenly chosen an inefficient plan, there is nothing to be done. The article [Kabra and DeWitt 1998] describes how a database system can monitor the execution of a plan, collecting statistics about its behavior. If it thinks that the execution is less efficient than it should be, it can use the statistics to create a better plan and "hot-swap" the old plan with the new one.

24.10 Exercises

CONCEPTUAL EXERCISES

24.1 Show that the *product* operator is associative.

24.2 Consider the query tree of Figure 24-2(a).
a) Give a sequence of transformations that will create the following tree:



b) Give a sequence of transformations that will create a left-deep tree having the join order (COURSE, SECTION, ENROLL, STUDENT, DEPT).

24.3 Consider a query that takes the product of several tables, and any two query trees equivalent to this query. Show that it is possible to use the equivalences of Section 24.1.1 to transform one tree into the other.

24.4 Consider a query tree containing a *select* node.
a) Show that moving the *select* node past another *select* node results in an equivalent query tree.
b) When can the *select* node be moved above a *project* node?
c) Show that a *select* node can be moved above or below a *groupby* node if meaningful to do so.

24.5 Consider the *union* relational algebra operator.
a) Show that the operator is associative and commutative, and give transformations for these equivalences.
b) Give a transformation that allows a selection to be pushed inside of a union.

24.6 Consider the *antijoin* and *semijoin* relational algebra operators.
a) Are these operators associative? Are they commutative? Give any appropriate transformations.
b) Give transformations that allow selections to be pushed inside of an antijoin or semijoin.

24.7 Consider adding the selection predicate of Figure 24-6(b) to the query tree of Figure 24-2(b). Give the query tree that results from pushing the selections as far as possible.

24.8 Give two equivalent query trees such that the lowest-cost plan comes from the higher-cost tree.

24.9 Show that the bushy tree of Figure 24-14(e) is the most promising tree for its SQL query.

24.10 The query tree of Figure 24-6(c) has the join order (STUDENT, ENROLL, SECTION, COURSE, DEPT). There are 15 other join orders that do not require *product* operations. Enumerate them.

24.11 Give a query such that Heuristic 5 does not produce the lowest-cost query tree.

24.12 Consider Figure 24-6.
a) Calculate the cost of each of the two trees.
b) Calculate the most promising query tree, using the heuristic algorithm.
c) Calculate the most promising query tree, using the dynamic programming algorithm.
d) Calculate the lowest-cost plan for the most promising query tree.

24.13 Consider the following query:

```
select Grade
from ENROLL, STUDENT, SECTION
where SId=StudentId and SectId=SectionId
and   SId = 1 and SectId=53
```

a) Show that the join order (ENROLL, STUDENT, SECTION) has a lower cost tree than (ENROLL, SECTION, STUDENT).
b) Calculate the most promising query tree, using the heuristic algorithm.
c) Calculate the most promising query tree, using the dynamic programming algorithm.
d) Calculate the lowest-cost plan for the most promising query tree.

24.14 The dynamic programming algorithm given in Section 24.4 only considers left-deep trees. Extend it to consider all possible join orders.

PROGRAMMING EXERCISES

24.15 Revise the SimpleDB heuristic planner so that it uses Heuristic 6b to choose the tables in the join order.

24.16 Implement a Selinger-style query planner for SimpleDB.

24.17 Exercise 23.15 asked you to implement the hashjoin algorithm in SimpleDB. Now modify the class *TablePlanner* so that it creates a hashjoin plan instead of a multibuffer product, when possible.

REFERENCES

Ailamaki, A., DeWitt, D., and Hill, M. (2002) *Data page layouts for relational databases on deep memory hierarchies*. VLDB Journal 11:3, pp. 198-215.

Alur, D., Crupi, J., and Malks, D. (2001) *Core J2EE Patterns: Best Practices and Design Strategies*. Pearson Education. An online version of the book can be found at *java.sun.com/blueprints/corej2eepatterns/index.html*.

Atkinson, M., Daynes, L., Jordan, M., Printezis, T., and Spence, S. (1996) *An orthogonally persistent Java.* ACM SIGMOD Record 25:4, pp. 68-75.

Astrahan, M., Blasgen, M., Chamberlin, D., Eswaren, K., Gray, J., Griffiths, P., King, W., Lorie, R., McJones, P., Mehl, J., Putzolu, G., Traiger, I., Wade, B., and Watson, V. (1976) *System R: Relational Approach to Database Management*. ACM Transactions on Database Systems 1:2, pp. 97-137.

Atzeni, P. and DeAntonellis, V. (1992) *Relational Database Theory*. Prentice-Hall.

Batory, D. and Gotlieb, C. (1982) *A Unifying Model of Physical Databases*. ACM Transactions on Database Systems 7:4, pp. 509-539.

Bello, R., Dias, K., Downing, A., Feenan, J., Finnerty, J., Norcott, W., Sun, H., Witkowski, A., and Ziauddin, M. (1998) *Materialized Views in Oracle*. Proceedings of the VLDB Conference, pp. 659-664.

Bayer, R. and Schkolnick, M. (1977) *Concurrency of operations on B-trees*. Acta Informatica 9:1, pp. 1-21.

Bayer, R. and Unterauer, K. (1977) *Prefix B-trees*. ACM Transactions on Database Systems 2:1, pp. 11-26.

Bernstein, P., Hadzilacos, V., and Goodman, N. (1987) *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Bernstein, P. and Newcomer, E. (1997) *Principles of Transaction Processing*. Morgan Kaufman.

Bowman, J. (2001) *Practical SQL: The Sequel*. Addison-Wesley.

Bruno, N. and Chaudhuri, S. (2004) *Conditional Selectivity for Statistics on Query Expressions*. Proceedings of the ACM SIGMOD Conference, pp. 311-322.

Carey, M., DeWitt, D., Richardson, J. and Shekita, E. (1986) *Object and File Management in the EXODUS Extendable Database System*. Proceedings of the VLDB Conference, pp. 91-100.

Castano, S., Fugini, M., Martella, G., and Samarati, P. (1995) *Database Security*. ACM Press (Addison-Wesley).

Celo, J. (1999) *SQL for Smarties: Advanced SQL Programming*. Morgan Kaufman.

Ceri, S., Cochrane, R., and Widom, J. (2000) *Practical Applications of Triggers and Constraints: Success and Lingering Issues*. Proceedings of the VLDB Conference, pp. 254-262.

Chaudhuri, S. (1998) *An Overview of Query Optimization in Relational Systems*. Proceedings of the ACM Principles of Database Systems Conference, pp. 34-43.

Chen, P., Lee, E., Gibson, G., and Patterson, D. (1995) *RAID: High-Performance, Reliable Secondary Storage*. ACM Computing Surveys 26:2, pp. 145-185.

Chen, S., Gibbons, P., Mowry, T., and Valentin, G. (2002) *Fractal Prefetching B^+-Trees: Optimizing Both Cache and Disk Performance*. Proceedings of the ACM SIGMOD Conference, pp. 157-168.

Copeland, G. and Maier, D. (1984) *Making Smalltalk a Database System.* Proceedings of the ACM SIGMOD Conference, pp. 316-325.

Date, C. (1986) *Relational Database: Selected Writings*. Addison Wesley.

Date, C. and Darwen, H. (2004) *A Guide to the SQL Standard (4th edition)*. Addison Wesley.

Date, C. and Darwen, H. (2006) *Databases, Types, and the Relational Model*. Addison Wesley.

Deitel, H., Deitel, P., Goldberg, A. (2003) *Internet & World Wide Web How to Program, 3rd edition*. Prentice Hall.

Dietrich, S. and Urban, S. (2004) *An Advanced Course in Database Systems: Beyond Relational Databases*. Prentice-Hall.

Effelsberg, W., and Haerder, T. (1984) *Principles of Database Buffer Management.* ACM Transactions on Database Systems 9:4, pp. 560-595.

Faloutsos, C. (1985) *Access Methods for Text*. ACM Computing Surveys 17:1, pp. 49-74.

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., and Shasha, D. (2005) *Making Snapshot Isolation Serializable*. ACM Transactions on Database Systems 30:2, pp. 492-528.

Fisher, M., Ellis, J., and Bruce, J. (2003) *JDBC API Tutorial and Reference (3rd edition)*. Addison Wesley.

Fowler, M. (2003) *Patterns of Enterprise Application Architecture*. Addison Wesley.

Fowler, M. and Scott, K. (2003) *UML Distilled*. Addison-Wesley.

Gibbons, P., Matias, Y., and Poosala, V. (2002) *Fast Incremental Maintenance of Incremental Histograms*. ACM Transactions on Database Systems 27:3, pp. 261-298.

Graede, V. and Gunther, O. (1998) *Multidimensional Access Methods*. ACM Computing Surveys 30:2, pp. 170-231.

Graefe, G. (1993) *Query Evaluation Techniques for Large Databases*. ACM Computing Surveys 25:2, pp. 73-170.

Graefe, G. (ed.) (1993) *Database Engineering Bulletin: Special Issue on Query Processing in Commercial Database Systems* (sites.computer.org/debull/93DEC-CD.pdf).

Graefe, G. (2003) *Sorting and Indexing With Partitioned B-Trees*. Proceedings of the CIDR Conference.

Graefe, G. (2004) *Write-Optimized B-Trees*. Proceedings of the VLDB Conference, pp. 672-683.

Graefe, G. (2006) *Implementing Sorting in Database Systems*. ACM Computing Surveys 38:3, pp. 1-37.

Gray, J. and Reuter, A. (1993) *Transaction Processing Concepts and Techniques*. Morgan Kaufman.

Grosso, W. (2001) *Java RMI*. O'Reilly.

Gulutzan, P. and Pelzer, T. (1999) *SQL-99 Complete, Really*. R&D Books.

Gupta, A. and Mumick, I. (eds.) (1999) *Materialized Views: Techniques, Implementations, and Applications*. MIT Press.

Haigh, T. (2006) *"A Veritable Bucket of Facts". Origins of the Data Base Management System.* ACM SIGMOD RECORD 35:2, pp. 33-49.

Hall, M. and Brown, L. (2004) *Core Servlets and JavaServer Pages: Volume I: Core Technologies, 2nd edition*. Prentice Hall.

Hay, D. (1996) *Data Model Patterns*. Dorsett House.

Hernandez, M. (2003) *Database Design for Mere Mortals*. Addison-Wesley.

Jannert, P. (2003) *Practical Database Design*, *Part 2*. IBM DeveloperWorks Library (www.ibm.com/developerworks/web/library/wa-dbdsgn2.html).

Kabra, N. and DeWitt, D. (1998) *Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans*. Proceedings of the ACM SIGMOD Conference, pp. 106-117.

Keith, M. and Schincariol, M. (2006) *Pro EJB 3: Java Persistence API*. Apress.

Kent, W. (2000) *Data and Reality*. 1st Books Library.

Knuth, D. (1998) *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.

Kreines, D. (2003) *Oracle Data Dictionary Pocket Reference*. O'Reilly.

Larman, C. (2005) *Applying UML and Patterns, 3rd edition*. Prentice Hall.
Lightstone, S., Teorey, T. and Nadeau, T. (2007) *Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more, 4th edition*. Morgan Kaurmann.

Lee, S. and Moon, B. (2007) *Design of Flash-Based DBMS: An In-Page Logging Approach*. Proceedings of the ACM-SIGMOD Conference, pp. 55-66.

Lomet, D. (2001) *The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account*. ACM SIGMOD Record 30:3, pp. 64-69.

Mangano, S. (2005) *XSLT Cookbook, 2nd edition*. O'Reilly.

Matias, Y., Vitter, J. and Wang, M. (1998) *Wavelet-Based Histograms for Selectivity Estimation*. Proceedings of the ACM SIGMOD Conference, pp. 448-459.

McHugh, J., Abiteboul, S., Goldman, R., Quass, D., and Widom, J. (1997) *Lore: A Database Management System for Semistructured Data*. SIGMOD Record 26(3), pp. 54-66 (www-db.stanford.edu/lore/pubs/lore97.pdf).

McLaughlin, B. (2006) *Java and XML, 3rd edition*. O'Reilly.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwartz, P. (1992) *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.* ACM Transactions on Database Systems 17:1, pp. 94-162.

Moss, J. (1985) *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press.

Nagar, R. (1997) *Windows NT File System Internals*. O'Reilly.

Nanda, N. (2002) *Drivers in the Wild*. JavaWorld (www.javaworld.net/javaworld/jw-07-2000/jw-0707-jdbc.html).

Nanda, N. and Kumar, S. (2002) *Create Your Own Type 3 JDBC Driver*. JavaWorld (www.javaworld.com/javaworld/jw-05-2002/jw-0517-jdbcdriver.html).

National Research Council committee on Innovations in Computing and Communications (1999) *Funding a Revolution*. National Academy Press.

Ng, R., Faloutsos, C., and Sellis, T. (1991). *Flexible Buffer Allocation Based on Marginal Gains*. Proceedings of the ACM SIGMOD Conference, pp. 387-396.

O'Neil, P. and Quass, D. (1997) *Improved Query Performance with Variant Indexes*. Proceedings of the ACM SIGMOD Conference, pp. 38-49.

Orfali, R., Harkey, D., and Edwards, J. (1999) *Client/Server Survival Guide (3rd edition).* J Wiley and Sons.

Pirahesh, H., Hellerstein, J., and Hasan, W. (1992) *Extendable/Rule Based Query Rewrite in Starburst*. Proceedings of the ACM SIGMOD Conference, pp. 39-48.

Rosenfeld, L. and Morville, P. (2006) *Information Architecture for the World Wide Web: Designing Large-Scale Web Sites, 3rd edition*. O'Reilly.

Sasha, D. and Bonnet, P. (2002) *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann.

Sciore, E. (1980) *The Universal Instance and Database Design*. Technical Report TR-271, Princeton University.

Scott, M. (2000) *Programming Language Pragmatics*. Morgan Kaufman.

Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T. (1979) Access-path selection in a relational database management system. Proceedings of the ACM SIGMOD Conference, pp. 23-34.

Shapiro, L. (1986) *Join Processing in Database Systems with Large Main Memories*. ACM Transactions on Database Systems 11:3, pp. 239-264.

Sieg, J, and Sciore, E. (1990) *Extended Relations*. Proceedings of the IEEE Data Engineering Conference, pp. 488-494.

Silberschatz, A., Gagne, G., and Galvin, P. (2004) *Operating System Concepts*. Addison Wesley.

Stahlbery, P., Miklau, G., and Levine, B. (2007) *Threats to Privacy in the Forensic Analysis of Database Systems*. Proceedings of the ACM SIGMOD Conference, pp. 91-102.

Stonebraker, M., Kreps, P., Wong, E., and Held, G. (1976) *The Design and Implementation of INGRES*. ACM Transactions on Database Systems 1:3, pp. 189-222.

Stonebraker, M., Abadi, D., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., and Zdonik, S. (2005) *C-Store: A Column-oriented DBMS*. Proceedings of the VLDB Conference, pp. 553-564.

Swami, A. (1989) *Optimization of large join queries: combining heuristics and combinatorial techniques*. ACM SIGMOD Record 18:2, pp. 367-376.

Teorey, T. (1999) *Database Modeling and Design: The E-R Approach*. Morgan Kaufman.

Vakali, A., Catania, B., and Maddalena, A. (2005) *XML Data Stores: Emerging Practices*. IEEE Internet Computing 9:2, pp. 62-69.

van Otegem, M. (2002) *Teach Yourself XSLT in 21 Days*. SAMS.

von Hagen, W. (2002) *Linux Filesystems*. Sams publishing.

Weikum, G. (1991) *Principles and Realization Strategies of Multilevel Transaction Management*. ACM Transactions on Database Systems 16:1, pp. 132-180.

Widom, J. and Ceri, S. (1996) *Active Database Systems*. Morgan Kaufman.

Wu, C. and Kuo, T. (2006) *The Design of Efficient Initialization and Crash Recovery for Log-based File Systems Over Flash Memory*. ACM Transactions on Storage 2:4, pp. 449-467.

Yu, P. and Cornell, D. (1993) *Buffer Management Based on Return on Consumption in a Multi-Query Environment*. VLDB Journal 2:1, pp. 1-37.

Zhou, J., Larson, P., and Elmongui, H. (2007) *Lazy Maintenance of Materialized Views*. Proceedings of the VLDB Conference, pp. 231-242.