# Computer Science

## Volume 1

Silberschatz–Korth–Sudarshan • *Database System Concepts, Fourth Edition*

querying textual data, including hyperlink-based techniques used in Web search engines.

Chapter 23 covers advanced data types and new applications, including temporal data, spatial and geographic data, multimedia data, and issues in the management of mobile and personal databases. Finally, Chapter 24 deals with advanced transaction processing. We discuss transaction-processing monitors, high-performance transaction systems, real-time transaction systems, and transactional workflows.

- **Case studies** (Chapters 25 through 27). In this part we present case studies of three leading commercial database systems, including Oracle, IBM DB2, and Microsoft SQL Server. These chapters outline unique features of each of these products, and describe their internal structure. They provide a wealth of interesting information about the respective products, and help you see how the various implementation techniques described in earlier parts are used in real systems. They also cover several interesting practical aspects in the design of real systems.
- **Online appendices**. Although most new database applications use either the relational model or the object-oriented model, the network and hierarchical data models are still in use. For the benefit of readers who wish to learn about these data models, we provide appendices describing the network and hierarchical data models in Appendices A and B respectively; the appendices are available only online (http://www.bell-labs.com/topic/books/db-book).

  Appendix C describes advanced relational database design, including the theory of multivalued dependencies, join dependencies, and the project-join and domain-key normal forms. This appendix is for the benefit of individuals who wish to cover the theory of relational database design in more detail, and instructors who wish to do so in their courses. This appendix, too, is available only online, on the Web page of the book.



## The Fourth Edition

The production of this fourth edition has been guided by the many comments and suggestions we received concerning the earlier editions, by our own observations while teaching at IIT Bombay, and by our analysis of the directions in which database technology is evolving.

Our basic procedure was to rewrite the material in each chapter, bringing the older material up to date, adding discussions on recent developments in database technology, and improving descriptions of topics that students found difficult to understand. Each chapter now has a list of review terms, which can help you review key topics covered in the chapter. We have also added a tools section at the end of most chapters, which provide information on software tools related to the topic of the chapter. We have also added new exercises, and updated references.

We have added a new chapter covering XML, and three case study chapters covering the leading commercial database systems, including Oracle, IBM DB2, and Microsoft SQL Server.

ers, such as programming exercises, project suggestions, online labs and tutorials, and teaching tips.

E-mail should be addressed to db-book@research.bell-labs.com. Any other correspondence should be sent to Avi Silberschatz, Bell Laboratories, Room 2T-310, 600 Mountain Avenue, Murray Hill, NJ 07974, USA.

## Acknowledgments

This edition has benefited from the many useful comments provided to us by the numerous students who have used the third edition. In addition, many people have written or spoken to us about the book, and have offered suggestions and comments. Although we cannot mention all these people here, we especially thank the following:

- Phil Bernhard, Florida Institute of Technology; Eitan M. Gurari, The Ohio State University; Irwin Levinstein, Old Dominion University; Ling Liu, Georgia Institute of Technology; Ami Motro, George Mason University; Bhagirath Narahari, Meral Ozsoyoglu, Case Western Reserve University; and Odinaldo Rodriguez, King's College London; who served as reviewers of the book and whose comments helped us greatly in formulating this fourth edition.
- Soumen Chakrabarti, Shirish Mehrotra, Krithi Ramamritham, Mike Reiter, Sunita Sarawagi, S. Sarda, and Dilys Thomas, for extensive and invaluable feedback on several chapters of the book.
- Phil Bohannon, for writing the first draft of Chapter 10 describing XML.
- Hakan Jakobsson (Oracle), Sriram Padmanabhan (IBM), and César Galindo-Legaria, Goetz Graefe, José A. Blakeley, Kalen Delaney, Michael Rys, Michael Zwilling, Sameet Agarwal, Thomas Casey (all of Microsoft) for writing the appendices describing the Oracle, IBM DB2, and Microsoft SQL Server database systems.
- Yuri Breitbart, for help with the distributed database chapter; Mike Reiter, for help with the security sections; and Jim Melton, for clarifications on SQL:1999.
- Marilyn Turnamian and Nandprasad Joshi, whose excellent secretarial assistance was essential for timely completion of this fourth edition.

The publisher was Betsy Jones. The senior developmental editor was Kelley Butcher. The project manager was Jill Peter. The executive marketing manager was John Wannemacher. The cover illustrator was Paul Tumbaugh while the cover designer was JoAnne Schopler. The freelance copyeditor was George Watson. The freelance proofreader was Marie Zartman. The supplement producer was Jodi Banowetz. The designer was Rick Noel. The freelance indexer was Tobiah Waldron.

This edition is based on the three previous editions, so we thank once again the many people who helped us with the first three editions, including R. B. Abhyankar, Don Batory, Haran Boral, Paul Bourgeois, Robert Brazile, Michael Carey, J. Edwards, Christos Faloutsos, Homma Farian, Alan Fekete, Shashi Gadia, Jim Gray, Le Gruen-

If the above query were run on the tables in Figure 1.3, the system would find that the two accounts numbered A-101 and A-201 are owned by customer 192-83-7465 and would print out the balances of the two accounts, namely 500 and 900.

There are a number of database query languages in use, either commercially or experimentally. We study the most widely used query language, SQL, in Chapter 4. We also study some other query languages in Chapter 5.

The levels of abstraction that we discussed in Section 1.3 apply not only to defining or structuring data, but also to manipulating data. At the physical level, we must define algorithms that allow efficient access to data. At higher levels of abstraction, we emphasize ease of use. The goal is to allow humans to interact efficiently with the system. The query processor component of the database system (which we study in Chapters 13 and 14) translates DML queries into sequences of actions at the physical level of the database system.



### 1.5.3 Database Access from Application Programs

**Application programs** are programs that are used to interact with the database. Application programs are usually written in a *host* language such as Cobol, C, C++, or Java. Examples in a banking system are programs that generate payroll checks, debit accounts, credit accounts, or transfer funds between accounts.

To access the database, DML statements need to be executed from the host language. There are two ways to do this:

- By providing an application program interface (set of procedures) that can be used to send DML and DDL statements to the database, and retrieve the results.

  The Open Database Connectivity (ODBC) standard defined by Microsoft for use with the C language is a commonly used application program interface standard. The Java Database Connectivity (JDBC) standard provides corresponding features to the Java language.

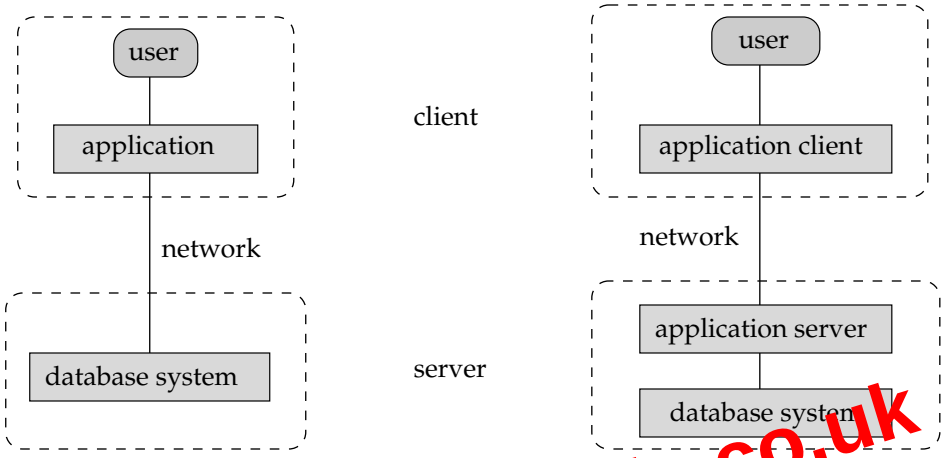
- By extending the host language syntax to embed DML calls within the host language program. Usually, a special character prefaces DML calls, and a preprocessor, called the *DML* **precompiler**, converts the DML statements to normal procedure calls in the host language.

## 1.6 Database Users and Administrators

A primary goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users or database administrators.

### 1.6.1 Database Users and User Interfaces

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.




a. two-tier architecture b. three-tier architecture

**Figure 1.5** two-tier and three-tier architectures.



writing data to a new tape. Data could also be input from punched card decks, and output to printers. For example, salary raises were processed by entering the raises on punched cards and reading the punched card deck in synchronization with a tape containing the master salary details. The records had to be in the same sorted order. The salary raises would be added to the salary read from the master tape, and written to a new tape; the new tape would become the new master tape.

Tapes (and card decks) could be read only sequentially, and data sizes were much larger than main memory; thus, data processing programs were forced to process data in a particular order, by reading and merging data from tapes and card decks.

- **Late 1960s and 1970s**: Widespread use of hard disks in the late 1960s changed the scenario for data processing greatly, since hard disks allowed direct access to data. The position of data on disk was immaterial, since any location on disk could be accessed in just tens of milliseconds. Data were thus freed from the tyranny of sequentiality. With disks, network and hierarchical databases could be created that allowed data structures such as lists and trees to be stored on disk. Programmers could construct and manipulate these data structures.

  A landmark paper by Codd [1970] defined the relational model, and nonprocedural ways of querying data in the relational model, and relational databases were born. The simplicity of the relational model and the possibility of hiding implementation details completely from the programmer were enticing indeed. Codd later won the prestigious Association of Computing Machinery Turing Award for his work.

PART 1

# Data Models



A **data model** is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. In this part, we study two data models—the entity–relationship model and the relational model.

The entity–relationship (E-R) model is a high-level data model. It is based on a perception of a real world that consists of a collection of basic objects, called *entities*, and of *relationships* among these objects.

The relational model is a lower-level model. It uses a collection of tables to represent both data and the relationships among those data. Its conceptual simplicity has led to its widespread adoption; today a vast majority of database products are based on the relational model. Designers often formulate database schema design by first modeling data at a high level, using the E-R model, and then translating it into the the relational model.

We shall study other data models later in the book. The object-oriented data model, for example, extends the representation of entities by adding notions of encapsulation, methods (functions), and object identity. The object-relational data model combines features of the object-oriented data model and the relational data model. Chapters 8 and 9, respectively, cover these two data models.

| | | | |
|---|---|---|---|
| 321-12-3123 | Jones | Main | Harrison |
| 019-28-3746 | Smith | North | Rye |
| 677-89-9011 | Hayes | Main | Harrison |
| 555-55-5555 | Jackson | Dupont | Woodside |
| 244-66-8800 | Curry | North | Rye |
| 963-96-3963 | Williams | Nassau | Princeton |
| 335-57-7991 | Adams | Spring | Pittsfield |

*customer*

| | |
|---|---|
| L-17 | 1000 |
| L-23 | 2000 |
| L-15 | 1500 |
| L-14 | 1500 |
| L-19 | 500 |
| L-11 | 900 |
| L-16 | 1300 |

*loan*

**Figure 2.1** Entity sets *customer* and *loan*.



- **Simple** and **composite** attributes. In our examples thus far, the attributes have been simple; that is, they are not divided into subparts. **Composite** attributes, on the other hand, can be divided into subparts (that is, other attributes). For example, an attribute *name* could be structured as a composite attribute consisting of *first-name*, *middle-initial*, and *last-name*. Using composite attributes in a design schema is a good choice if a user will wish to refer to an entire attribute on some occasions, and to only a component of the attribute on other occasions. Suppose we were to substitute for the *customer* entity-set attributes *customer-street* and *customer-city* the composite attribute *address* with the attributes *street*, *city*, *state*, and *zip-code*.[2] Composite attributes help us to group together related attributes, making the modeling cleaner.

  Note also that a composite attribute may appear as a hierarchy. In the composite attribute *address*, its component attribute *street* can be further divided into *street-number*, *street-name*, and *apartment-number*. Figure 2.2 depicts these examples of composite attributes for the *customer* entity set.

- **Single-valued** and **multivalued** attributes. The attributes in our examples all have a single value for a particular entity. For instance, the *loan-number* attribute for a specific loan entity refers to only one loan number. Such attributes are said to be **single valued**. There may be instances where an attribute has a set of values for a specific entity. Consider an *employee* entity set with the attribute *phone-number*. An employee may have zero, one, or several phone numbers, and different employees may have different numbers of phones. This type of attribute is said to be **multivalued**. As another example, an at-

2. We assume the address format used in the United States, which includes a numeric postal code called a zip code.

entities. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed as relationships.

### 2.4.3 Binary versus n-ary Relationship Sets

Relationships in databases are often binary. Some relationships that appear to be nonbinary could actually be better represented by several binary relationships. For instance, one could create a ternary relationship *parent*, relating a child to his/her mother and father. However, such a relationship could also be represented by two binary relationships, *mother* and *father*, relating a child to his/her mother and father separately. Using the two relationships *mother* and *father* allows us record a child's mother, even if we are not aware of the father's identity; a null value would be required if the ternary relationship *parent* is used. Using binary relationship sets is preferable in this case.

In fact, it is always possible to replace a nonbinary (n-ary, for $n > 2$) relationship set by a number of distinct binary relationship sets. For simplicity, consider the abstract ternary ($n = 3$) relationship set $R$, relating entity sets $A$, $B$, and $C$. We replace the relationship set $R$ by an entity set $E$, and create three relationship sets:

- $R_A$, relating $E$ and $A$
- $R_B$, relating $E$ and $B$
- $R_C$, relating $E$ and $C$



If the relationship set $R$ had any attributes, these are assigned to entity set $E$; further, a special identifying attribute is created for $E$ (since it must be possible to distinguish different entities in an entity set on the basis of their attribute values). For each relationship $(a_i, b_i, c_i)$ in the relationship set $R$, we create a new entity $e_i$ in the entity set $E$. Then, in each of the three new relationship sets, we insert a relationship as follows:

- $(e_i, a_i)$ in $R_A$
- $(e_i, b_i)$ in $R_B$
- $(e_i, c_i)$ in $R_C$

We can generalize this process in a straightforward manner to $n$-ary relationship sets. Thus, conceptually, we can restrict the E-R model to include only binary relationship sets. However, this restriction is not always desirable.

- An identifying attribute may have to be created for the entity set created to represent the relationship set. This attribute, along with the extra relationship sets required, increases the complexity of the design and (as we shall see in Section 2.9) overall storage requirements.
- A $n$-ary relationship set shows more clearly that several entities participate in a single relationship.

- An undirected line from the relationship set *borrower* to the entity set *loan* specifies that *borrower* is either a many-to-many or one-to-many relationship set from *customer* to *loan*.

Returning to the E-R diagram of Figure 2.8, we see that the relationship set *borrower* is many-to-many. If the relationship set *borrower* were one-to-many, from *customer* to *loan*, then the line from *borrower* to *customer* would be directed, with an arrow pointing to the *customer* entity set (Figure 2.9a). Similarly, if the relationship set *borrower* were many-to-one from *customer* to *loan*, then the line from *borrower* to *loan* would have an arrow pointing to the *loan* entity set (Figure 2.9b). Finally, if the relationship set *borrower* were one-to-one, then both lines from *borrower* would have arrows:

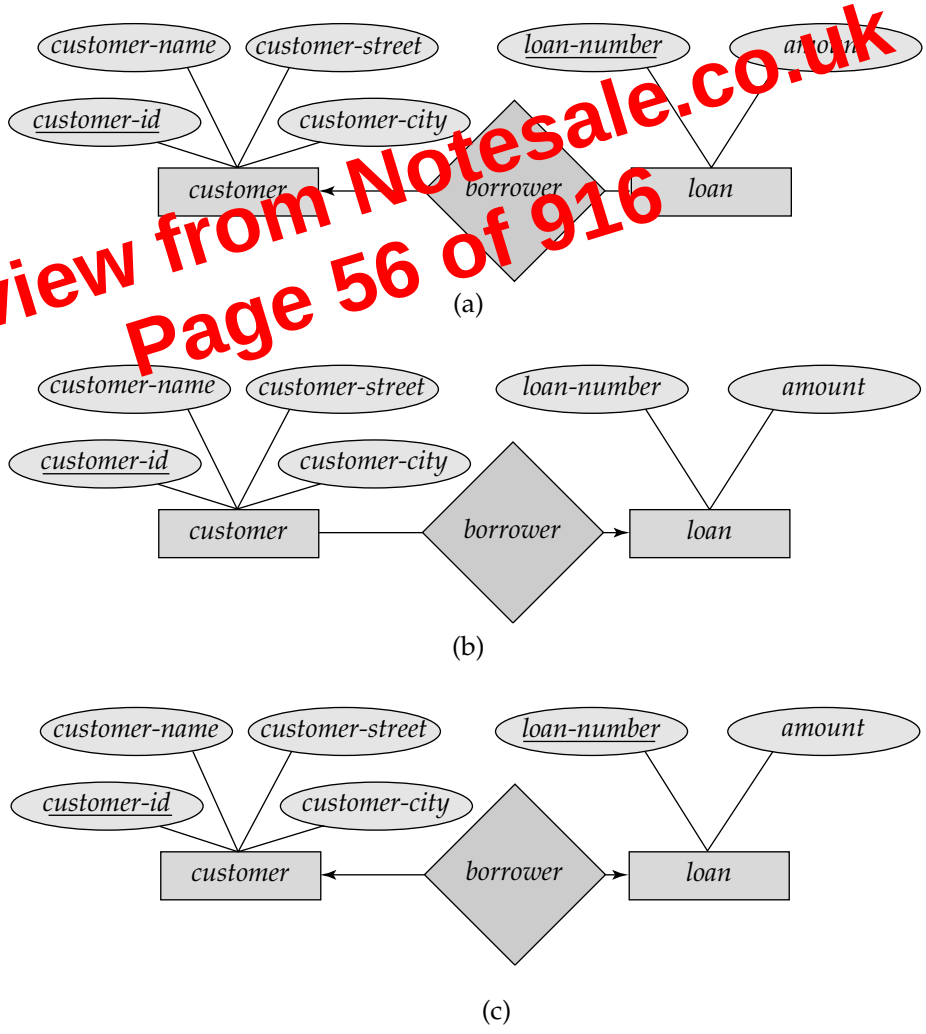





**Figure 2.9** Relationships. (a) one to many. (b) many to one. (c) one-to-one.

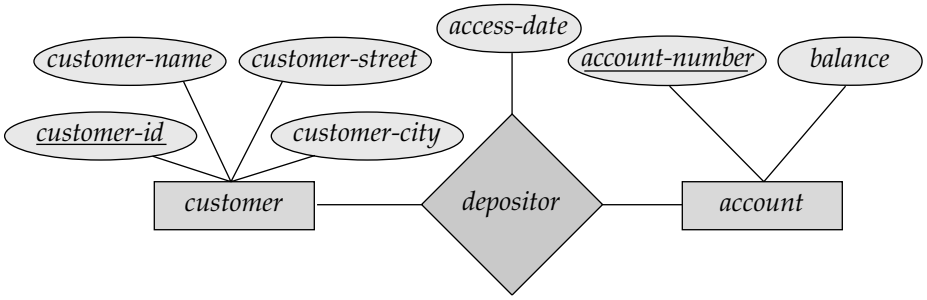



**Figure 2.10** E-R diagram with an attribute attached to a relationship set.

one pointing to the *loan* entity set and one pointing to the *customer* entity set (Figure 2.9c).

If a relationship set has also some attributes associated with it, then we link these attributes to that relationship set. For example, in Figure 2.10, we have the *access-date* descriptive attribute attached to the relationship set *depositor* to specify the most recent date on which a customer accessed that account.

Figure 2.11 shows how composite attributes can be represented in the E-R notation. Here, a composite attribute *name*, with component attributes *first-name*, *middle-initial*, and *last-name* replaces the simple attribute *customer-name* of *customer*. Also, a composite attribute *address*, whose component attributes are *street*, *city*, *state*, and *zip-code* replaces the attributes *customer-street* and *customer-city* of *customer*. The attribute *street* is itself a composite attribute whose component attributes are *street-number*, *street-name*, and *apartment number*.

Figure 2.11 also illustrates a multivalued attribute *phone-number*, depicted by a double ellipse, and a derived attribute *age*, depicted by a dashed ellipse.

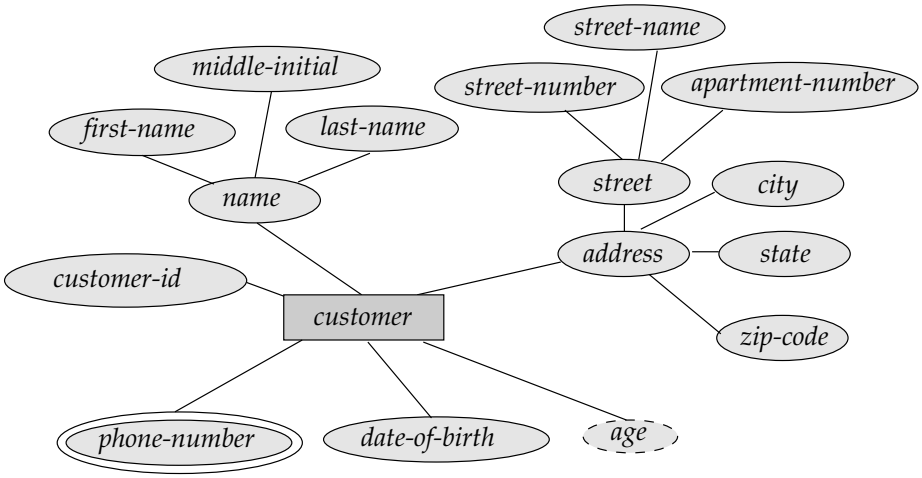



**Figure 2.11** E-R diagram with composite, multivalued, and derived attributes.

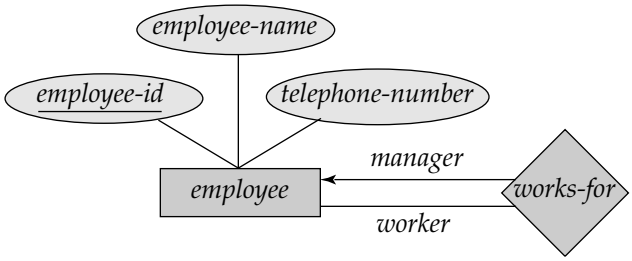



**Figure 2.12** E-R diagram with role indicators.

We indicate roles in E-R diagrams by labeling the lines that connect diamonds to rectangles. Figure 2.12 shows the role indicators *manager* and *worker* between the *employee* entity set and the *works-for* relationship set.

Nonbinary relationship sets can be specified easily in an E-R diagram. Figure 2.13 consists of the three entity sets *employee*, *job*, and *branch*, related through the relationship set *works-on*.

We can specify some types of many-to-one relationships in the case of nonbinary relationship sets. Suppose an employee can have at most one job in each branch (for example, jobs cannot be a manager and an auditor at the same branch). This constraint can be specified by an arrow pointing to *job* on the edge from *works-on*.

We permit at most one arrow out of a relationship set, since an E-R diagram with two or more arrows out of a nonbinary relationship set can be interpreted in two ways. Suppose there is a relationship set $R$ between entity sets $A_1, A_2, \ldots, A_n$, and the only arrows are on the edges to entity sets $A_{i+1}, A_{i+2}, \ldots, A_n$. Then, the two possible interpretations are:

1. A particular combination of entities from $A_1, A_2, \ldots, A_i$ can be associated with at most one combination of entities from $A_{i+1}, A_{i+2}, \ldots, A_n$. Thus, the primary key for the relationship $R$ can be constructed by the union of the primary keys of $A_1, A_2, \ldots, A_i$.

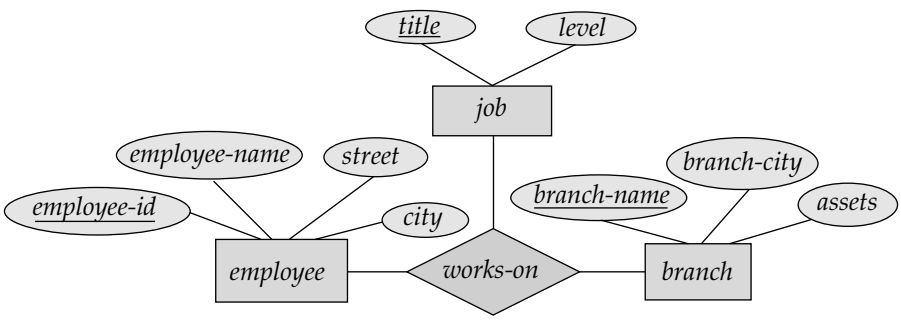



**Figure 2.13** E-R diagram with a ternary relationship.

- Two account entity sets—*savings-account* and *checking-account*—with the common attributes of *account-number* and *balance*; in addition, *savings-account* has the attribute *interest-rate* and *checking-account* has the attribute *overdraft-amount*.
- The *loan* entity set, with the attributes *loan-number*, *amount*, and *originating-branch*.
- The weak entity set *loan-payment*, with attributes *payment-number*, *payment-date*, and *payment-amount*.

### 2.8.2.3 Relationship Sets Designation

We now return to the rudimentary design scheme of Section 2.8.2.2 and specify the following relationship sets and mapping cardinalities. In the process, we also refine some of the decisions we made earlier regarding attributes of entity sets.

- *borrower*, a many-to-many relationship set between *customer* and *loan*.
- *loan-branch*, a many-to-one relationship set that indicates in which branch a loan originated. Note that this relationship set replaces the attribute *originating-branch* of the entity set *loan*.
- *loan-payment*, a one-to-many relationship from *loan* to *payment*, which documents that a payment is made on a loan.
- *depositor*, with relationship attribute *access-date*, a many-to-many relationship set between *customer* and *account*, indicating that a customer owns an account.
- *cust-banker*, with relationship attribute *type*, a many-to-one relationship set expressing that a customer can be advised by a bank employee, and that a bank employee can advise one or more customers. Note that this relationship set has replaced the attribute *banker-name* of the entity set *customer*.
- *works-for*, a relationship set between *employee* entities with role indicators *manager* and *worker*; the mapping cardinalities express that an employee works for only one manager and that a manager supervises one or more employees. Note that this relationship set has replaced the *manager* attribute of *employee*.



### 2.8.2.4 E-R Diagram

Drawing on the discussions in Section 2.8.2.3, we now present the completed E-R diagram for our example banking enterprise. Figure 2.22 depicts the full representation of a conceptual model of a bank, expressed in terms of E-R concepts. The diagram includes the entity sets, attributes, relationship sets, and mapping cardinalities arrived at through the design processes of Sections 2.8.2.1 and 2.8.2.2, and refined in Section 2.8.2.3.

mal superkey for each relationship set from among its superkeys; this is the relationship set's primary key.

- An entity set that does not have sufficient attributes to form a primary key is termed a **weak entity set**. An entity set that has a primary key is termed a **strong entity set**.
- **Specialization** and **generalization** define a containment relationship between a higher-level entity set and one or more lower-level entity sets. Specialization is the result of taking a subset of a higher-level entity set to form a lower-level entity set. Generalization is the result of taking the union of two or more disjoint (lower-level) entity sets to produce a higher-level entity set. The attributes of higher-level entity sets are inherited by lower-level entity sets.
- **Aggregation** is an abstraction in which relationship sets (along with their associated entity sets) are treated as higher-level entity sets, and can participate in relationships.
- The various features of the E-R model offer the database designer numerous choices in how to best represent the enterprise being modeled. Concepts and objects may, in certain cases, be represented by entities, relationships, or attributes. Aspects of the overall structure of the enterprise may be best described by using weak entity sets, generalization, specialization, or aggregation. Often, the designer must weigh the merits of a simple, compact model versus those of a more precise, but more complex, one.
- A database that conforms to an E-R diagram can be represented by a collection of tables. For each entity set and for each relationship set in the database, there is a unique table that is assigned the name of the corresponding entity set or relationship set. Each table has a number of columns, each of which has a unique name. Converting database representation from an E-R diagram to a table format is the basis for deriving a relational-database design from an E-R diagram.
- The **unified modeling language (UML)** provides a graphical means of modeling various components of a software system. The class diagram component of UML is based on E-R diagrams. However, there are some differences between the two that one must beware of.



## Review Terms

- Entity-relationship data model
- Entity
- Entity set
- Attributes
- Domain
- Simple and composite attributes
- Single-valued and multivalued attributes
- Null value
- Derived attribute
- Relationship, and relationship set
- Role

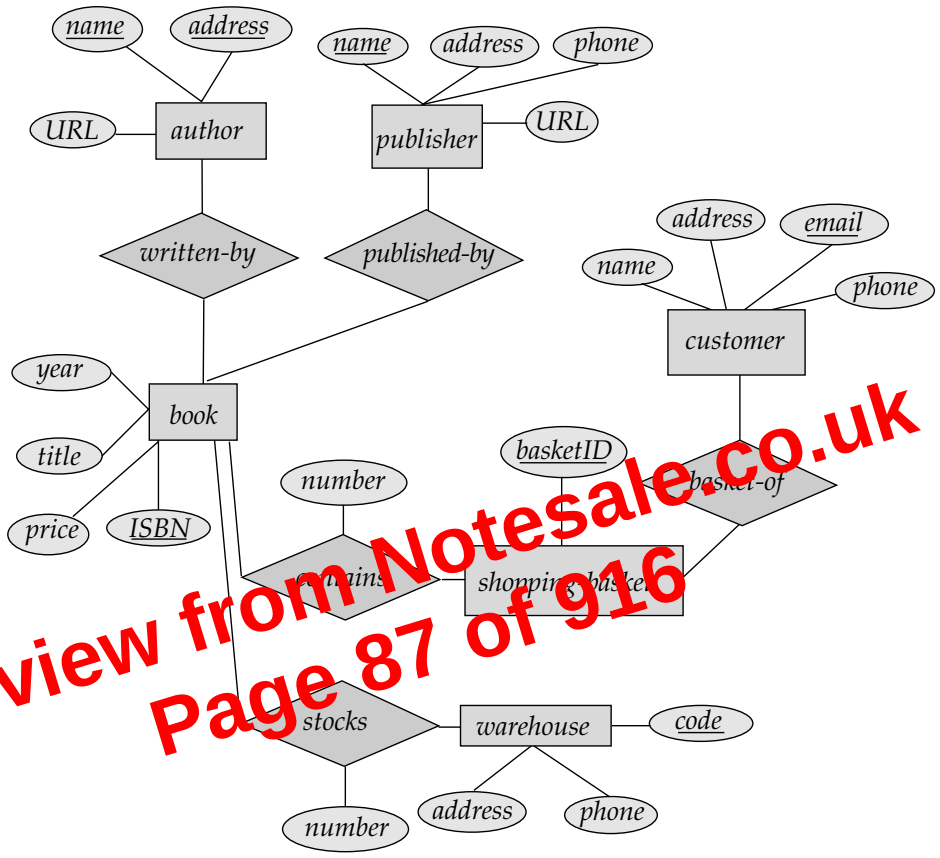





**Figure 2.29** E-R diagram for Exercise 2.12.

**b.** Explain what application characteristics would influence a decision to include or not to include each of the additional entity sets.

**2.15** When designing an E-R diagram for a particular enterprise, you have several alternatives from which to choose.

**a.** What criteria should you consider in making the appropriate choice?
**b.** Design three alternative E-R diagrams to represent the university registrar's office of Exercise 2.4. List the merits of each. Argue in favor of one of the alternatives.

**2.16** An E-R diagram can be viewed as a graph. What do the following mean in terms of the structure of an enterprise schema?

**a.** The graph is disconnected.
**b.** The graph is acyclic.

**2.17** In Section 2.4.3, we represented a ternary relationship (Figure 2.30a) using binary relationships, as shown in Figure 2.30b. Consider the alternative shown in

In a real-world database, the *customer-id* (which could be a *social-security* number, or an identifier generated by the bank) would serve to uniquely identify customers.

We also need a relation to describe the association between customers and accounts. The relation schema to describe this association is

$$Depositor\text{-}schema = (customer\text{-}name, account\text{-}number)$$

Figure 3.5 shows a sample relation *depositor* (*Depositor-schema*).

It would appear that, for our banking example, we could have just one relation schema, rather than several. That is, it may be easier for a user to think in terms of one relation schema, rather than in terms of several. Suppose that we used only one relation for our example, with schema

(*branch-name*, *branch-city*, *assets*, *customer-name*, *customer-street*, *customer-city*, *account-number*, *balance*)



Observe that, if a customer has several accounts, we must list her address once for each account. That is, we must repeat certain information several times. This repetition is wasteful and is avoided by the use of several relations, as in our example.

In addition, if a branch has no accounts (a newly created branch, say, that has no customers yet), we cannot construct a complete tuple on the preceding single relation, because no data concerning *customer* and *account* are available yet. To represent incomplete tuples, we must use *null* values that signify that the value is unknown or does not exist. Thus, in our example, the values for *customer-name*, *customer-street*, and so on must be null. By using several relations, we can represent the branch information for a bank with no customers without using null values. We simply use a tuple on *Branch-schema* to represent the information about the branch, and create tuples on the other schemas only when the appropriate information becomes available.

In Chapter 7, we shall study criteria to help us decide when one set of relation schemas is more appropriate than another, in terms of information repetition and the existence of null values. For now, we shall assume that the relation schemas are given.

We include two additional relations to describe data about loans maintained in the various branches in the bank:

| *customer-name* | *account-number* |
|---|---|
| Hayes | A-102 |
| Johnson | A-101 |
| Johnson | A-201 |
| Jones | A-217 |
| Lindsay | A-222 |
| Smith | A-215 |
| Turner | A-305 |

**Figure 3.5** The *depositor* relation.

| *customer-name* | *borrower. loan-number* | *loan. loan-number* | *branch-name* | *amount* |
|---|---|---|---|---|
| Adams | L-16 | L-15 | Perryridge | 1500 |
| Adams | L-16 | L-16 | Perryridge | 1300 |
| Curry | L-93 | L-15 | Perryridge | 1500 |
| Curry | L-93 | L-16 | Perryridge | 1300 |
| Hayes | L-15 | L-15 | Perryridge | 1500 |
| Hayes | L-15 | L-16 | Perryridge | 1300 |
| Jackson | L-14 | L-15 | Perryridge | 1500 |
| Jackson | L-14 | L-16 | Perryridge | 1300 |
| Jones | L-17 | L-15 | Perryridge | 1500 |
| Jones | L-17 | L-16 | Perryridge | 1300 |
| Smith | L-11 | L-15 | Perryridge | 1500 |
| Smith | L-11 | L-16 | Perryridge | 1300 |
| Smith | L-23 | L-15 | Perryridge | 1500 |
| Smith | L-23 | L-16 | Perryridge | 1300 |
| Williams | L-17 | L-15 | Perryridge | 1500 |
| Williams | L-17 | L-16 | Perryridge | 1300 |

**Figure 3.15** Result of $\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \times loan)$.



who do not have a loan at the Perryridge branch. (If you do not see why that is true, recall that the Cartesian product takes all possible pairings of one tuple from *borrower* with one tuple of *loan*.)

Since the Cartesian-product operation associates *every* tuple of *loan* with every tuple of *borrower*, we know that, if a customer has a loan in the Perryridge branch, then there is some tuple in *borrower* × *loan* that contains his name, and *borrower.loan-number* = *loan.loan-number*. So, if we write

$$\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \times loan))$$

we get only those tuples of *borrower* × *loan* that pertain to customers who have a loan at the Perryridge branch.

Finally, since we want only *customer-name*, we do a projection:

$$\Pi_{customer\text{-}name}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \times loan)))$$

The result of this expression, shown in Figure 3.16, is the correct answer to our query.

### 3.2.1.7 The Rename Operation

Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful to be able to give them names; the **rename** operator, denoted by the lowercase Greek letter rho ($\rho$), lets us do

| *customer-name* |
|---|
| Adams |
| Hayes |

**Figure 3.16** Result of $\Pi_{customer\text{-}name}$ $(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}$ $(\sigma_{branch\text{-}name = \text{“Perryridge”}}$ $(borrower \times loan)))$.

this. Given a relational-algebra expression $E$, the expression

$$\rho_x (E)$$

returns the result of expression $E$ under the name $x$.

A relation $r$ by itself is considered a (trivial) relational-algebra expression. Thus, we can also apply the rename operation to a relation $r$ to get the same relation under a new name.

A second form of the rename operation is as follows. Assume that a relational-algebra expression $E$ has arity $n$. Then, the expression

$$\rho_{x(A_1, A_2, \ldots, A_n)} (E)$$

returns the result of expression $E$ under the name $x$, and with the attributes renamed to $A_1, A_2, \ldots, A_n$.



To illustrate renaming a relation, we consider the query “Find the largest account balance in the bank.” Our strategy is to (1) compute first a temporary relation consisting of those balances that are *not* the largest and (2) take the set difference between the relation $\Pi_{balance} (account)$ and the temporary relation just computed, to obtain the result.

Step 1: To compute the temporary relation, we need to compare the values of all account balances. We do this comparison by computing the Cartesian product $account \times account$ and forming a selection to compare the value of any two balances appearing in one tuple. First, we need to devise a mechanism to distinguish between the two *balance* attributes. We shall use the rename operation to rename one reference to the account relation; thus we can reference the relation twice without ambiguity.

| *balance* |
|---|
| 500 |
| 400 |
| 700 |
| 750 |
| 350 |

**Figure 3.17** Result of the subexpression $\Pi_{account.balance}$ $(\sigma_{account.balance < d.balance}$ $(account \times \rho_d (account)))$.

| *balance* |
|---|
| 900 |

**Figure 3.18** Largest account balance in the bank.

We can now write the temporary relation that consists of the balances that are not the largest:

$$\Pi_{account.balance} \left(\sigma_{account.balance < d.balance} \left(account \times \rho_d \left(account\right)\right)\right)$$

This expression gives those balances in the *account* relation for which a larger balance appears somewhere in the *account* relation (renamed as *d*). The result contains all balances *except* the largest one. Figure 3.17 shows this relation.

Step 2: The query to find the largest account balance in the bank can be written as:

$$\Pi_{balance} \left(account\right) - \Pi_{account.balance} \left(\sigma_{account.balance < d.balance} \left(account \times \rho_d \left(account\right)\right)\right)$$

Figure 3.18 shows the result of this query.

As one more example of the rename operation, consider the query "Find the names of all customers who live on the same street and in the same city as Smith." We can obtain Smith's street and city by writing

$$\Pi_{customer\text{-}street,\ customer\text{-}city} \left(\sigma_{customer\text{-}name\ =\ \text{“Smith”}} \left(customer\right)\right)$$

However, in order to find other customers with this street and city, we must reference the *customer* relation a second time. In the following query, we use the rename operation on the preceding expression to give its result the name *smith-addr*, and to rename its attributes to *street* and *city*, instead of *customer-street* and *customer-city*:

$$\Pi_{customer.customer\text{-}name} \left(\sigma_{customer.customer\text{-}street = smith\text{-}addr.street \wedge customer.customer\text{-}city = smith\text{-}addr.city} \left(customer \times \rho_{smith\text{-}addr(street, city)} \left(\Pi_{customer\text{-}street,\ customer\text{-}city} \left(\sigma_{customer\text{-}name\ =\ \text{“Smith”}} \left(customer\right)\right)\right)\right)\right)$$

The result of this query, when we apply it to the *customer* relation of Figure 3.4, appears in Figure 3.19.

The rename operation is not strictly required, since it is possible to use a positional notation for attributes. We can name attributes of a relation implicitly by using a positional notation, where $1, $2, . . . refer to the first attribute, the second attribute, and so on. The positional notation also applies to results of relational-algebra operations.

| *customer-name* |
|---|
| Curry |
| Smith |

**Figure 3.19** Customers who live on the same street and in the same city as Smith.

### 3.2.3.1 The Set-Intersection Operation

The first additional-relational algebra operation that we shall define is **set intersection** ($\cap$). Suppose that we wish to find all customers who have both a loan and an account. Using set intersection, we can write

$$\Pi_{customer\text{-}name}\ (borrower)\ \cap\ \Pi_{customer\text{-}name}\ (depositor)$$

The result relation for this query appears in Figure 3.20.

Note that we can rewrite any relational algebra expression that uses set intersection by replacing the intersection operation with a pair of set-difference operations as:

$$r \cap s = r - (r - s)$$

Thus, set intersection is not a fundamental operation and does not add any power to the relational algebra. It is simply more convenient to write $r \cap s$ than to write $r - (r - s)$.



### 3.2.3.2 The Natural-Join Operation

It is often desirable to simplify certain queries that require a Cartesian product. Usually, a query that involves a Cartesian product includes a selection operation on the result of the Cartesian product. Consider the query "Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount." We first form the Cartesian product of the *borrower* and *loan* relations. Then, we select those tuples that pertain to only the same *loan-number*, followed by the projection of the resulting *customer-name*, *loan-number*, and *amount*:

$$\Pi_{customer\text{-}name,\ loan.loan\text{-}number,\ amount}\ (\sigma_{borrower.loan\text{-}number\ =\ loan.loan\text{-}number}\ (borrower\ \times\ loan))$$

The *natural join* is a binary operation that allows us to combine certain selections and a Cartesian product into one operation. It is denoted by the "join" symbol $\bowtie$. The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.

Although the definition of natural join is complicated, the operation is easy to apply. As an illustration, consider again the example "Find the names of all customers who have a loan at the bank, and find the amount of the loan." We express this query

| *customer-name* |
|---|
| Hayes |
| Jones |
| Smith |

**Figure 3.20** Customers with both an account and a loan at the bank.

There are cases where we must eliminate multiple occurrences of a value before computing an aggregate function. If we do want to eliminate duplicates, we use the same function names as before, with the addition of the hyphenated string "**distinct**" appended to the end of the function name (for example, **count-distinct**). An example arises in the query "Find the number of branches appearing in the *pt-works* relation." In this case, a branch name counts only once, regardless of the number of employees working that branch. We write this query as follows:

$$\mathcal{G}_{\textbf{count-distinct}(branch\text{-}name)}(pt\text{-}works)$$

For the relation in Figure 3.27, the result of this query is a single row containing the value 3.

Suppose we want to find the total salary sum of all part-time employees at each branch of the bank separately, rather than the sum for the entire bank. To do so, we need to partition the relation *pt-works* into **groups** based on the branch, and to apply the aggregate function on each group.

The following expression using the aggregation operator $\mathcal{G}$ achieves the desired result:

$$_{branch\text{-}name}\mathcal{G}_{\textbf{sum}(salary)}(pt\text{-}works)$$

In the expression, the attribute *branch-name* in the left-hand subscript of $\mathcal{G}$ indicates that the input relation *pt-works* must be divided into groups based on the value of *branch-name*. Figure 3.28 shows the resulting groups. The expression $\textbf{sum}(salary)$ in the right-hand subscript of $\mathcal{G}$ indicates that for each group of tuples (that is, each branch), the aggregation function **sum** must be applied on the collection of values of the *salary* attribute. The output relation consists of tuples with the branch name, and the sum of the salaries for the branch, as shown in Figure 3.29.

The general form of the **aggregation operation** $\mathcal{G}$ is as follows:

$$_{G_1,G_2,\ldots,G_n}\mathcal{G}_{F_1(A_1),\ F_2(A_2),\ldots,\ F_m(A_m)}(E)$$

where $E$ is any relational-algebra expression; $G_1, G_2, \ldots, G_n$ constitute a list of attributes on which to group; each $F_i$ is an aggregate function; and each $A_i$ is an at-

| *employee-name* | *branch-name* | *salary* |
|---|---|---|
| Rao | Austin | 1500 |
| Sato | Austin | 1600 |
| Johnson | Downtown | 1500 |
| Loreena | Downtown | 1300 |
| Peterson | Downtown | 2500 |
| Adams | Perryridge | 1500 |
| Brown | Perryridge | 1300 |
| Gopal | Perryridge | 5300 |

**Figure 3.28** The *pt-works* relation after grouping.

Instead of specifying a tuple as we did earlier, we specify a set of tuples that is inserted into both the *account* and *depositor* relation. Each tuple in the *account* relation has an *account-number* (which is the same as the loan number), a *branch-name* (Perryridge), and the initial balance of the new account ($200). Each tuple in the *depositor* relation has as *customer-name* the name of the loan customer who is being given the new account and the same account number as the corresponding *account* tuple.

### 3.4.3 Updating

In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple. We can use the generalized-projection operator to do this task:

$$r \leftarrow \Pi_{F_1, F_2, \ldots, F_n}(r)$$

where each $F_i$ is either the $i$th attribute of $r$, if the $i$th attribute is not updated, or, if the attribute is to be updated, $F_i$ is an expression, involving only constants and the attributes of $r$, that gives the new value for the attribute.

If we want to select some tuples from $r$ and to update only them, we can use the following expression; here, $P$ denotes the selection condition that chooses which tuples to update:

$$r \leftarrow \Pi_{F_1, F_2, \ldots, F_n}(\sigma_P(r)) \cup (r - \sigma_P(r))$$

To illustrate the use of the update operation, suppose that interest payments are being made, and that all balances are to be increased by 5 percent. We write

$$account \leftarrow \Pi_{account\text{-}number,\ branch\text{-}name,\ balance\ *1.05}\ (account)$$

Now suppose that accounts with balances over $10,000 receive 6 percent interest, whereas all others receive 5 percent. We write

$$\begin{aligned} account \leftarrow\ & \Pi_{AN, BN,\ balance\ *1.06}\ (\sigma_{balance > 10000}\ (account)) \\ & \cup\ \Pi_{AN,\ BN\ balance\ *1.05}\ (\sigma_{balance \leq 10000}\ (account)) \end{aligned}$$

where the abbreviations $AN$ and $BN$ stand for *account-number* and *branch-name*, respectively.

## 3.5 Views

In our examples up to this point, we have operated at the logical-model level. That is, we have assumed that the relations in thecollection we are given are the actual relations stored in the database.

It is not desirable for all users to see the entire logical model. Security considerations may require that certain data be hidden from users. Consider a person who needs to know a customer's loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in the relational algebra, by

$$\Pi_{customer\text{-}name,\ loan\text{-}number,\ branch\text{-}name}\ (borrower \bowtie\ loan)$$

Aside from security concerns, we may wish to create a personalized collection of relations that is better matched to a certain user's intuition than is the logical model.

$P$ or that appear in one or more relations whose names appear in $P$. For example, $dom(t \in loan \wedge t[amount] > 1200)$ is the set containing 1200 as well as the set of all values appearing in *loan*. Also, $dom(\neg (t \in loan))$ is the set of all values appearing in *loan*, since the relation $loan$ is mentioned in the expression.

We say that an expression $\{t \mid P(t)\}$ is *safe* if all values that appear in the result are values from $dom(P)$. The expression $\{t \mid \neg (t \in loan)\}$ is not safe. Note that $dom(\neg (t \in loan))$ is the set of all values appearing in *loan*. However, it is possible to have a tuple *t* not in *loan* that contains values that do not appear in *loan*. The other examples of tuple-relational-calculus expressions that we have written in this section are safe.

### 3.6.4 Expressive Power of Languages

The tuple relational calculus restricted to safe expressions is equivalent in expressive power to the basic relational algebra (with the operators $\cup$, $-$, $\times$, $\sigma$, and $\rho$, but without the extended relational operators such as generalized projection $\mathcal{G}$ and the outer-join operations) Thus, for every relational-algebra expression using only the basic operations, there is an equivalent expression in the tuple relational calculus, and for every tuple-relational-calculus expression, there is an equivalent relational-algebra expression. We will not prove this assertion here; the bibliographic notes contain references to the proof. Some parts of the proof are included in the exercises. We note that the tuple relational calculus does not have any equivalent of the aggregate operation, but it can be extended to support aggregation. Extending the tuple relational calculus to handle arithmetic expressions is straightforward.



## 3.7 The Domain Relational Calculus**

A second form of relational calculus, called **domain relational calculus**, uses *domain* variables that take on values from an attributes domain, rather than values for an entire tuple. The domain relational calculus, however, is closely related to the tuple relational calculus.

Domain relational calculus serves as the theoretical basis of the widely used QBE language, just as relational algebra serves as the basis for the SQL language.

### 3.7.1 Formal Definition

An expression in the domain relational calculus is of the form

$$\{< x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

where $x_1, x_2, \ldots, x_n$ represent domain variables. $P$ represents a formula composed of atoms, as was the case in the tuple relational calculus. An atom in the domain relational calculus has one of the following forms:

- $< x_1, x_2, \ldots, x_n > \in r$, where *r* is a relation on *n* attributes and $x_1, x_2, \ldots, x_n$ are domain variables or domain constants.

We note that the domain relational calculus also does not have any equivalent of the aggregate operation, but it can be extended to support aggregation, and extending it to handle arithmatic expressions is straightforward.

## 3.8 Summary

- The **relational data model** is based on a collection of tables. The user of the database system may query these tables, insert new tuples, delete tuples, and update (modify) tuples. There are several languages for expressing these operations.
- The **relational algebra** defines a set of algebraic operations that operate on tables, and output tables as their results. These operations can be combined to get expressions that express desired queries. The algebra defines the basic operations used within relational query languages.
- The operations in relational algebra can be divided into
  - Basic operations
  - Additional operations that can be expressed in terms of the basic operations
  - Extended operations, some of which add further expressive power to relational algebra
- Databases can be modified by **insertion**, **deletion**, or **update** of tuples. We used the relational algebra with the **assignment operator** to express these modifications.
- Different users of a shared database may benefit from individualized **views** of the database. Views are "virtual relations" defined by a query expression. We evaluate queries involving views by replacing the view with the expression that defines the view.
- Views are useful mechanisms for simplifying database queries, but modification of the database through views may cause problems. Therefore, database systems severely restrict updates through views.
- For reasons of query-processing efficiency, a view may be **materialized**—that is, the query is evaluated and the result stored physically. When database relations are updated, the materialized view must be correspondingly updated.
- The **tuple relational calculus** and the **domain relational calculus** are nonprocedural languages that represent the basic power required in a relational query language. The basic relational algebra is a procedural language that is equivalent in power to both forms of the relational calculus when they are restricted to safe expressions.
- The relational algebra and the relational calculi are terse, formal languages that are inappropriate for casual users of a database system. Commercial database systems, therefore, use languages with more "syntactic sugar." In Chap-

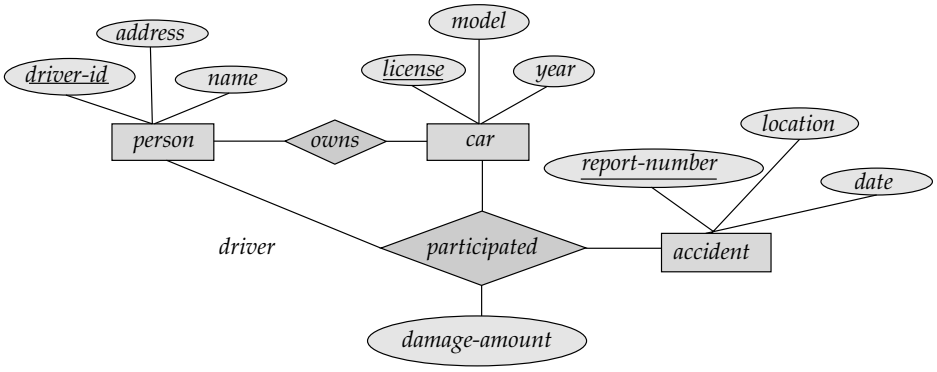



**Figure 3.38** E-R diagram.



**3.2** Describe the differences in meaning between the terms *relation* and *relation schema*. Illustrate your answer by referring to your solution to Exercise 3.1.

**3.3** Design a relational database corresponding to the E-R diagram of Figure 3.38.

**3.4** In Chapter 2 we saw how to represent many-to-many, many-to-one, one-to-many, and one-to-one relationship sets. Explain how primary keys help us to represent such relationship sets in the relational model.

**3.5** Consider the relational database of Figure 3.39, where the primary keys are underlined. Give an expression in the relational algebra to express each of the following queries:

- **a.** Find the names of all employees who work for First Bank Corporation.
- **b.** Find the names and cities of residence of all employees who work for First Bank Corporation.
- **c.** Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000 per annum.
- **d.** Find the names of all employees in this database who live in the same city as the company for which they work.
- **e.** Find the names of all employees who live in the same city and on the same street as do their managers.
- **f.** Find the names of all employees in this database who do not work for First Bank Corporation.
- **g.** Find the names of all employees who earn more than every employee of Small Bank Corporation.
- **h.** Assume the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.

**3.6** Consider the relation of Figure 3.21, which shows the result of the query "Find the names of all customers who have a loan at the bank." Rewrite the query to include not only the name, but also the city of residence for each customer. Observe that now customer Jackson no longer appears in the result, even though Jackson does in fact have a loan from the bank.

projects the result onto the attributes of the **select** clause. In practice, SQL may convert the expression into an equivalent form that can be processed more efficiently. However, we shall defer concerns about efficiency to Chapters 13 and 14.

### 4.2.1 The select Clause

The result of an SQL query is, of course, a relation. Let us consider a simple query using our banking example, "Find the names of all branches in the *loan* relation":

**select** *branch-name*
**from** *loan*

The result is a relation consisting of a single attribute with the heading *branch-name*.

Formal query languages are based on the mathematical notion of a relation being a set. Thus, duplicate tuples never appear in relations. In practice, duplicate elimination is time-consuming. Therefore, SQL (like most other commercial query languages) allows duplicates in relations as well as in the results of SQL expressions. Thus, the preceding query will list each *branch-name* once for every tuple in which it appears in the *loan* relation.

In those cases where we want to force the elimination of duplicates, we insert the keyword **distinct** after **select**. We can rewrite the preceding query as

**select distinct** *branch-name*
**from** *loan*



if we want duplicates removed.

SQL allows us to use the keyword **all** to specify explicitly that duplicates are not removed:

**select all** *branch-name*
**from** *loan*

Since duplicate retention is the default, we will not use **all** in our examples. To ensure the elimination of duplicates in the results of our example queries, we will use **distinct** whenever it is necessary. In most queries where **distinct** is not used, the exact number of duplicate copies of each tuple present in the query result is not important. However, the number is important in certain applications; we return to this issue in Section 4.2.8.

The asterisk symbol " * " can be used to denote "all attributes." Thus, the use of *loan*.* in the preceding **select** clause would indicate that all attributes of *loan* are to be selected. A select clause of the form **select** * indicates that all attributes of all relations appearing in the **from** clause are selected.

The **select** clause may also contain arithmetic expressions involving the operators $+$, $-$, $*$, and / operating on constants or attributes of tuples. For example, the query

**select** *loan-number*, *branch-name*, *amount* * 100
**from** *loan*

Then $\Pi_B(r_1)$ would be $\{(a), (a)\}$, whereas $\Pi_B(r_1) \times r_2$ would be

$$\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$$

We can now define how many copies of each tuple occur in the result of an SQL query. An SQL query of the form

**select** $A_1,\ A_2, \ldots, A_n$
**from** $r_1,\ r_2, \ldots, r_m$
**where** $P$

is equivalent to the relational-algebra expression

$$\Pi_{A_1,\ A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \cdots \times r_m))$$

using the multiset versions of the relational operators $\sigma$, $\Pi$, and $\times$.



## 4.3 Set Operations

The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the relational-algebra operations $\cup$, $\cap$, and $-$. Like union, intersection, and set difference in relational algebra, the relations participating in the operations must be *compatible*; that is, they must have the same set of attributes.

Let us demonstrate how several of the example queries that we considered in Chapter 3 can be written in SQL. We shall now construct queries involving the **union**, **intersect**, and **except** operations of two sets: the set of all customers who have an account at the bank, which can be derived by

**select** *customer-name*
**from** *depositor*

and the set of customers who have a loan at the bank, which can be derived by

**select** *customer-name*
**from** *borrower*

We shall refer to the relations obtained as the result of the preceding queries as *d* and *b*, respectively.

### 4.3.1 The Union Operation

To find all customers having a loan, an account, or both at the bank, we write

(**select** *customer-name*
**from** *depositor*)
**union**
(**select** *customer-name*
**from** *borrower*)

The result of this query is a relation with a single attribute, containing a single tuple with a numerical value corresponding to the average balance at the Perryridge branch. Optionally, we can give a name to the attribute of the result relation by using the **as** clause.

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the **group by** clause. The attribute or attributes given in the **group by** clause are used to form groups. Tuples with the same value on all attributes in the **group by** clause are placed in one group.

As an illustration, consider the query "Find the average account balance at each branch." We write this query as follows:

**select** *branch-name*, **avg** (*balance*)
**from** *account*
**group by** *branch-name*

Retaining duplicates is important in computing an average. Suppose that the account balances at the (small) Brighton branch are \$1000, \$100, \$2000, and \$1000. The average balance is \$7000/4 = \$1750.00. If duplicates were eliminated, we would obtain the wrong answer (\$6000/3 = \$2000).

There are cases where we must eliminate duplicates before computing an aggregate function. If we do want to eliminate duplicates, we use the keyword **distinct** in the aggregate expression. An example arises in the query "Find the number of depositors for each branch." In this case, a depositor counts only once, regardless of the number of accounts that depositor may have. We write this query as follows:

**select** *branch-name*, **count** (**distinct** *customer-name*)
**from** *depositor, account*
**where** *depositor.account-number* = *account.account-number*
**group by** *branch-name*

At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those branches where the average account balance is more than \$1200. This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause. To express such a query, we use the **having** clause of SQL. SQL applies predicates in the **having** clause after groups have been formed, so aggregate functions may be used. We express this query in SQL as follows:

**select** *branch-name*, **avg** (*balance*)
**from** *account*
**group by** *branch-name*
**having avg** (*balance*) > 1200

At times, we wish to treat the entire relation as a single group. In such cases, we do not use a **group by** clause. Consider the query "Find the average balance for all accounts." We write this query as follows:

The result of an arithmetic expression (involving, for example $+$, $-$, $*$ or $/$) is null if any of the input values is null. SQL treats as **unknown** the result of any comparison involving a *null* value (other than **is null** and **is not null**).

Since the predicate in a **where** clause can involve Boolean operations such as **and**, **or**, and **not** on the results of comparisons, the definitions of the Boolean operations are extended to deal with the value **unknown**, as outlined in Section 3.3.4.

- **and**: The result of *true* **and** *unknown* is *unknown*, *false* **and** *unknown* is *false*, while *unknown* **and** *unknown* is *unknown*.
- **or**: The result of *true* **or** *unknown* is *true*, *false* **or** *unknown* is *unknown*, while *unknown* **or** *unknown* is *unknown*.
- **not**: The result of **not** *unknown* is *unknown*.

SQL defines the result of an SQL statement of the form

**select** . . . **from** $R_1, \ldots, R_n$ **where** $P$

to contain (projections of) tuples in $R_1 \times \cdots \times R_n$ for which predicate $P$ evaluates to **true**. If the predicate evaluates to either **false** or **unknown** for a tuple in $R_1 \times \cdots \times R_n$ (the projection of the tuple is not added to the result.

SQL also allows us to test whether the result of a comparison is unknown, rather than true or false, by using the clauses **is unknown** and **is not unknown**.

Null values, when they exist, also complicate the processing of aggregate operators. For example, assume that some tuples in the *loan* relation have a null value for *amount*. Consider the following query to total all loan amounts:

**select sum** (*amount*)
**from** *loan*

The values to be summed in the preceding query include null values, since some tuples have a null value for *amount*. Rather than say that the overall sum is itself *null*, the SQL standard says that the **sum** operator should ignore *null* values in its input.

In general, aggregate functions treat nulls according to the following rule: All aggregate functions except **count(*)** ignore null values in their input collection. As a result of null values being ignored, the collection of values may be empty. The **count** of an empty collection is defined to be 0, and all other aggregate operations return a value of null when applied on an empty collection. The effect of null values on some of the more complicated SQL constructs can be subtle.

A **boolean** type data, which can take values **true**, **false**, and **unknown**, was introduced in SQL:1999. The aggregate functions **some** and **every**, which mean exactly what you would intuitively expect, can be applied on a collection of Boolean values.

## 4.6 Nested Subqueries

SQL provides a mechanism for nesting subqueries. A subquery is a **select-from-where** expression that is nested within another query. A common use of subqueries

is to perform tests for set membership, make set comparisons, and determine set cardinality. We shall study these uses in subsequent sections.

### 4.6.1 Set Membership

SQL draws on the relational calculus for operations that allow testing tuples for membership in a relation. The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause. The **not in** connective tests for the absence of set membership. As an illustration, reconsider the query "Find all customers who have both a loan and an account at the bank." Earlier, we wrote such a query by intersecting two sets: the set of depositors at the bank, and the set of borrowers from the bank. We can take the alternative approach of finding all account holders at the bank who are members of the set of borrowers from the bank. Clearly, this formulation generates the same results as the previous one did, but it leads us to write our query using the **in** connective of SQL. We begin by finding all account holders, and we write the subquery

```
(select customer-name
 from depositor)
```

We then need to find those customers who are borrowers from the bank and who appear in the list of account holders obtained in the subquery. We do so by nesting the subquery in the outer **select**. The resulting query is

```
select distinct customer-name
from borrower
where customer-name in (select customer-name
                        from depositor)
```

This example shows that it is possible to write the same query several ways in SQL. This flexibility is beneficial, since it allows a user to think about the query in the way that seems most natural. We shall see that there is a substantial amount of redundancy in SQL.

In the preceding example, we tested membership in a one-attribute relation. It is also possible to test for membership in an arbitrary relation in SQL. We can thus write the query "Find all customers who have both an account and a loan at the Perryridge branch" in yet another way:

```
select distinct customer-name
from borrower, loan
where borrower.loan-number = loan.loan-number and
      branch-name = 'Perryridge' and
      (branch-name, customer-name) in
          (select branch-name, customer-name
           from depositor, account
           where depositor.account-number = account.account-number)
```

We must use the **close** statement to tell the database system to delete the temporary relation that held the result of the query. For our example, this statement takes the form

EXEC SQL **close** *c* END-EXEC

SQLJ, the Java embedding of SQL, provides a variation of the above scheme, where Java iterators are used in place of cursors. SQLJ associates the results of a query with an iterator, and the next() method of the Java iterator interface can be used to step through the result tuples, just as the preceding examples use **fetch** on the cursor.

Embedded SQL expressions for database modification (**update**, **insert**, and **delete**) do not return a result. Thus, they are somewhat simpler to express. A database-modification request takes the form

EXEC SQL < any valid **update, insert,** or **delete**> END-EXEC

Host-language variables, preceded by a colon, may appear in the SQL database-modification expression. If an error condition arises in the execution of the statement, a diagnostic is set in the SQLCA.

Database relations can also be updated through cursors. For example, if we want to add 100 to the *balance* attribute of every *account* where the branch name is "Perryridge", we could declare a cursor as follows.

**declare** *c* **cursor for**
**select** *
**from** *account*
**where** *branch-name* = 'Perryridge'
**for update**

We then iterate through the tuples by performing **fetch** operations on the cursor (as illustrated earlier), and after fetching each tuple we execute the following code

**update** *account*
**set** *balance* = *balance* + 100
**where current of** *c*

Embedded SQL allows a host-language program to access the database, but it provides no assistance in presenting results to the user or in generating reports. Most commercial database products include tools to assist application programmers in creating user interfaces and formatted reports. We discuss such tools in Chapter 5 (Section 5.3).

## 4.13 Dynamic SQL

The *dynamic SQL* component of SQL allows programs to construct and submit SQL queries at run time. In contrast, embedded SQL statements must be completely present at compile time; they are compiled by the embedded SQL preprocessor. Using dynamic SQL, programs can create SQL queries as strings at run time (perhaps based on

CHAPTER 5

# Other Relational Languages



In Chapter 4, we described SQL—the most influential commercial relational-database language. In this chapter, we study two more languages: QBE and Datalog. Unlike SQL, QBE is a graphical language, where queries *look* like tables. QBE and its variants are widely used in database systems on personal computers. Datalog has a syntax modeled after the Prolog language. Although not used commercially at present, Datalog has been used in several research database systems.

Here, we present fundamental constructs and concepts rather than a complete users' guide for these languages. Keep in mind that individual implementations of a language may differ in details, or may support only a subset of the full language.

In this chapter, we also study forms interfaces and tools for generating reports and analyzing data. While these are not strictly speaking languages, they form the main interface to a database for many users. In fact, most users do not perform explicit querying with a query language at all, and access data only via forms, reports, and other data analysis tools.

## 5.1 Query-by-Example

**Query-by-Example (QBE)** is the name of both a data-manipulation language and an early database system that included this language. The QBE database system was developed at IBM's T. J. Watson Research Center in the early 1970s. The QBE data-manipulation language was later used in IBM's Query Management Facility (QMF). Today, many database systems for personal computers support variants of QBE language. In this section, we consider only the data-manipulation language. It has two distinctive features:

1. Unlike most query languages and programming languages, QBE has a **two-dimensional syntax**: Queries *look* like tables. A query in a one-dimensional

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| | P._x | _y |
| | _x | ¬ _y |

In English, the preceding query reads "Display all *customer-name* values that appear in at least two tuples, with the second tuple having an *account-number* different from the first."

### 5.1.3 The Condition Box

At times, it is either inconvenient or impossible to express all the constraints on the domain variables within the skeleton tables. To overcome this difficulty, QBE includes a **condition box** feature that allows the expression of general constraints over any of the domain variables. QBE allows logical expressions to appear in a condition box. The logical operators are the words **and** and **or**, or the symbols "&" and "|".

For example, the query "Find the loan numbers of all loans made to Smith, to Jones (or to both jointly)" can be written as

| *borrower* | *customer-name* | *loan-number* |
|---|---|---|
| | _n | P._x |

| *conditions* |
|---|
| _n = Smith **or** _n = Jones |



It is possible to express the above query without using a condition box, by using P. in multiple rows. However, queries with P. in multiple rows are sometimes hard to understand, and are best avoided.

As yet another example, suppose that we modify the final query in Section 5.1.2 to be "Find all customers who are not named 'Jones' and who have at least two accounts." We want to include an "$x \neq$ Jones" constraint in this query. We do that by bringing up the condition box and entering the constraint "$x \neg =$ Jones":

| *conditions* |
|---|
| $x \neg =$ Jones |

Turning to another example, to find all account numbers with a balance between $1300 and $1500, we write

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| | P. | | _x |

| *conditions* |
|---|
| _x ≥ 1300 |
| _x ≤ 1500 |

As an illustration, consider the query "Find the *customer-name*, *account-number*, and *balance* for all accounts at the Perryridge branch." In relational algebra, we would construct this query as follows:

1. Join *depositor* and *account*.
2. Project *customer-name*, *account-number*, and *balance*.

To construct the same query in QBE, we proceed as follows:

1. Create a skeleton table, called *result*, with attributes *customer-name*, *account-number*, and *balance*. The name of the newly created skeleton table (that is, *result*) must be different from any of the previously existing database relation names.
2. Write the query.

The resulting query is

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| | _y | Perryridge | _z |

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| | _x | _y |

| *result* | *customer-name* | *account-number* | *balance* |
|---|---|---|---|
| P. | _x | _y | _z |



## 5.1.5 Ordering of the Display of Tuples

QBE offers the user control over the order in which tuples in a relation are displayed. We gain this control by inserting either the command AO. (ascending order) or the command DO. (descending order) in the appropriate column. Thus, to list in ascending alphabetic order all customers who have an account at the bank, we write

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| | P.AO. | |

QBE provides a mechanism for sorting and displaying data in multiple columns. We specify the order in which the sorting should be carried out by including, with each sort operator (AO or DO), an integer surrounded by parentheses. Thus, to list all account numbers at the Perryridge branch in ascending alphabetic order with their respective account balances in descending order, we write

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| | P.AO(1). | Perryridge | P.DO(2). |

| *depositor* | *customer-name* | *account-number* |
|---|---|---|
| | P.G._x | _y |

| *account* | *account-number* | *branch-name* | *balance* |
|---|---|---|---|
| | _y | _z | |

| *branch* | *branch-name* | *branch-city* | *assets* |
|---|---|---|---|
| | _z | Brooklyn | |
| | _w | Brooklyn | |

| *conditions* |
|---|
| CNT.UNQ._z = CNT.UNQ._w |

The domain variable $w$ can hold the value of names of branches located in Brooklyn. Thus, CNT.UNQ._$w$ is the number of distinct branches in Brooklyn. The domain variable $z$ can hold the value of branches in such a way that both of the following hold:

- The branch is located in Brooklyn.
- The customer whose name is $x$ has an account at the branch.

Thus, CNT.UNQ._$z$ is the number of distinct branches in Brooklyn at which customer $x$ has an account. If CNT.UNQ._$z$ = CNT.UNQ._$w$, then customer $x$ must have an account at all of the branches located in Brooklyn. In such a case, the displayed result includes $x$ (because of the P.).



## 5.1.7 Modification of the Database

In this section, we show how to add, remove, or change information in QBE.

### 5.1.7.1 Deletion

Deletion of tuples from a relation is expressed in much the same way as a query. The major difference is the use of D. in place of P. QBE (unlike SQL), lets us delete whole tuples, as well as values in selected columns. When we delete information in only some of the columns, null values, specified by −, are inserted.

We note that a D. command operates on only one relation. If we want to delete tuples from several relations, we must use one D. operator for each relation.

Here are some examples of QBE delete requests:

- Delete customer Smith.

| *customer* | *customer-name* | *customer-street* | *customer-city* |
|---|---|---|---|
| D. | Smith | | |

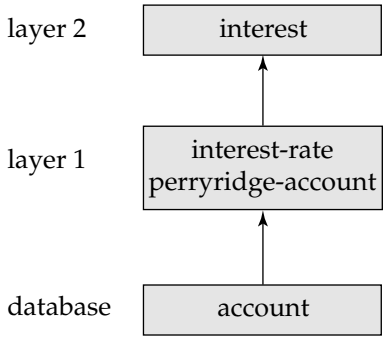



**Figure 5.9** Layering of view relations.



in level 1, since all the relations used in the two rules defining it are in the database. Relation *perryridge-account* is similarly in layer 1. Finally, relation *interest* is in layer 2, since it is not in layer 1 and all the relations used in the rule defining it are in the database or in layers lower than 2.

We can now define the semantics of a Datalog program in terms of the layering of view relations. Let the layers in a given program be $1, 2, \ldots, n$. Let $\mathcal{R}_i$ denote the set of all rules defining view relations in layer $i$.

- We define $I_0$ to be the set of facts stored in the database, and define $I_1$ as

$$I_1 = I_0 \cup infer(\mathcal{R}_1, I_0)$$

- We proceed in a similar fashion, defining $I_2$ in terms of $I_1$ and $\mathcal{R}_2$, and so on, using the following definition:

$$I_{i+1} = I_i \cup infer(\mathcal{R}_{i+1}, I_i)$$

- Finally, the set of facts in the view relations defined by the program (also called the **semantics of the program**) is given by the set of facts $I_n$ corresponding to the highest layer $n$.

For the program in Figure 5.6, $I_0$ is the set of facts in the database, and $I_1$ is the set of facts in the database along with all facts that we can infer from $I_0$ using the rules for relations *interest-rate* and *perryridge-account*. Finally, $I_2$ contains the facts in $I_1$ along with the facts for relation *interest* that we can infer from the facts in $I_1$ by the rule defining *interest*. The semantics of the program—that is, the set of those facts that are in each of the view relations—is defined as the set of facts $I_2$.

Recall that, in Section 3.5.3, we saw how to define the meaning of nonrecursive relational-algebra views by a technique known as view expansion. View expansion can be used with nonrecursive Datalog views as well; conversely, the layering technique described here can also be used with relational-algebra views.

ation is more complicated than using recursion, and evaluation by recursion can be optimized to run faster than evaluation by iteration.

The expressive power provided by recursion must be used with care. It is relatively easy to write recursive programs that will generate an infinite number of facts, as this program illustrates:

$$number(0)$$
$$number(A) \text{ :– } number(B), A = B + 1$$

The program generates $number(n)$ for all positive integers $n$, which is clearly infinite, and will not terminate. The second rule of the program does not satisfy the safety condition in Section 5.2.4. Programs that satisfy the safety condition will terminate, even if they are recursive, provided that all database relations are finite. For such programs, tuples in view relations can contain only constants from the database, and hence the view relations must be finite. The converse is not true; that is, there are programs that do not satisfy the safety conditions, but that do terminate.



### 5.2.8 Recursion in Other Languages

The SQL:1999 standard supports a limited form of recursion, using the **with recursive** clause. Suppose the relation *manager* has attributes *emp* and *mgr*. We can find every pair $(X, Y)$ such that $X$ is directly or indirectly managed by $Y$, using this SQL:1999 query:

```
with recursive empl(emp, mgr) as (
        select emp, mgr
        from manager
    union
        select emp, empl.mgr
        from manager, empl
        where manager.mgr = empl.emp
    )
select ∗
from empl
```

Recall that the **with** clause is used to define a temporary view whose definition is available only to the query where it is defined. The additional keyword **recursive** specifies that the view is recursive. The SQL definition of the view *empl* above is equivalent to the Datalog version we saw in Section 5.2.6.

The procedure Datalog-Fixpoint iteratively uses the function $infer(\mathcal{R}, I)$ to compute what facts are true, given a recursive Datalog program. Although we considered only the case of Datalog programs without negative literals, the procedure can also be used on views defined in other languages, such as SQL or relational algebra, provided that the views satisfy the conditions described next. Regardless of the language used to define a view $V$, the view can be thought of as being defined by an expression $E_V$ that, given a set of facts $I$, returns a set of facts $E_V(I)$ for the view relation $V$. Given a set of view definitions $\mathcal{R}$ (in any language), we can define a function

```
create domain HourlyWage numeric(5,2)
        constraint wage-value-test check(value >= 4.00)
```

The domain *HourlyWage* has a constraint that ensures that the hourly wage is greater than 4.00. The clause **constraint** *wage-value-test* is optional, and is used to give the name *wage-value-test* to the constraint. The name is used to indicate which constraint an update violated.

The **check** clause can also be used to restrict a domain to not contain any null values:

```
create domain AccountNumber char(10)
        constraint account-number-null-test check(value not null )
```

As another example, the domain can be restricted to contain only a specified set of values by using the **in** clause:

```
create domain AccountType char(10)
        constraint account-type-test
                check(value in ('Checking', 'Saving'))
```

The preceding **check** conditions can be tested quite easily, when a tuple is inserted or modified. However, in general, the **check** conditions can be more complex (and harder to check), since subqueries that refer to other relations are permitted in the **check** condition. For example, this constraint could be specified on the relation *deposit*:

```
check (branch-name in (select branch-name from branch))
```

The **check** condition verifies that the *branch-name* in each tuple in the *deposit* relation is actually the name of a branch in the *branch* relation. Thus, the condition has to be checked not only when a tuple is inserted or modified in *deposit*, but also when the relation *branch* changes (in this case, when a tuple is deleted or modified in relation *branch*).

The preceding constraint is actually an example of a class of constraints called *referential-integrity* constraints. We discuss such constraints, along with a simpler way of specifying them in SQL, in Section 6.2.

Complex **check** conditions can be useful when we want to ensure integrity of data, but we should use them with care, since they may be costly to test.

## 6.2 Referential Integrity

Often, we wish to ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called **referential integrity**.

wise been declared to be non-null. If all the columns of a foreign key are non-null in a given tuple, the usual definition of foreign-key constraints is used for that tuple. If any of the foreign-key columns is null, the tuple is defined automatically to satisfy the constraint.

This definition may not always be the right choice, so SQL also provides constructs that allow you to change the behavior with null values; we do not discuss the constructs here. To avoid such complexity, it is best to ensure that all columns of a **foreign key** specification are declared to be non-null.

Transactions may consist of several steps, and integrity constraints may be violated temporarily after one step, but a later step may remove the violation. For instance, suppose we have a relation *marriedperson* with primary key *name*, and an attribute *spouse*, and suppose that *spouse* is a foreign key on *marriedperson*. That is, the constraint says that the *spouse* attribute must contain a name that is present in the *person* table. Suppose we wish to note the fact that John and Mary are married to each other by inserting two tuples, one for John and one for Mary, to the above relation. The insertion of the first tuple would violate the foreign-key constraint, regardless of which of the two tuples is inserted first. After the second tuple is inserted the foreign key constraint would hold again.

To handle such situations, integrity constraints are checked at the end of a transaction, and not at intermediate steps.[1]



## 6.3 Assertions

An **assertion** is a predicate expressing a condition that we wish the database always to satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. We have paid substantial attention to these forms of assertion because they are easily tested and apply to a wide range of database applications. However, there are many constraints that we cannot express by using only these special forms. Two examples of such constraints are:

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.
- Every loan has at least one customer who maintains an account with a minimum balance of $1000.00.

An assertion in SQL takes the form

**create assertion** <assertion-name> **check** <predicate>

Here is how the two examples of constraints can be written. Since SQL does not provide a "for all $X$, $P(X)$" construct (where $P$ is a predicate), we are forced to im-

1. We can work around the problem in the above example in another way, if the *spouse* attribute can be set to null: We set the spouse attributes to null when inserting the tuples for John and Mary, and we update them later. However, this technique is rather messy, and does not work if the attributes cannot be set to null.

```
create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
   insert into borrower
          (select customer-name, account-number
           from depositor
           where nrow.account-number = depositor.account-number);
    insert into loan values
          (nrow.account-number, nrow.branch-name, − nrow.balance);
   update account set balance = 0
          where account.account-number = nrow.account-number
end
```

**Figure 6.3** Example of SQL:1999 syntax for triggers.



own syntax for triggers, leading to incompatibilities. We outline in Figure 6.3 the SQL:1999 syntax for triggers (which is similar to the syntax in the IBM DB2 and Oracle database systems).

This trigger definition specifies that the trigger is initiated *after* any update of the relation *account* is executed. An SQL update statement could update multiple tuples of the relation, and the **for each row** clause in the trigger code would then explicitly iterate over each updated row. The **referencing new row as** clause creates a variable *nrow* (called a **transition variable**), which stores the value of an updated row after the update.

The **when** statement specifies a condition, namely $nrow.balance < 0$. The system executes the rest of the trigger body only for tuples that satisfy the condition. The **begin atomic** ... **end** clause serves to collect multiple SQL statements into a single compound statement. The two **insert** statements with the **begin** ... **end** structure carry out the specific tasks of creating new tuples in the *borrower* and *loan* relations to represent the new loan. The **update** statement serves to set the account balance back to 0 from its earlier negative value.

The triggering event and actions can take many forms:

- The triggering *event* can be **insert** or **delete**, instead of **update**.

  For example, the action on **delete** of an account could be to check if the holders of the account have any remaining accounts, and if they do not, to delete them from the *depositor* relation. You can define this trigger as an exercise (Exercise 6.7).

  As another example, if a new *depositor* is inserted, the triggered action could be to send a welcome letter to the depositor. Obviously a trigger cannot directly cause such an action outside the database, but could instead add a tuple to a relation storing addresses to which welcome letters need to be sent. A separate process would go over this table, and print out letters to be sent.

```
create trigger overdraft-trigger on account
for update
as
if nrow.balance < 0
begin
   insert into borrower
          (select customer-name, account-number
           from depositor, inserted
           where inserted.account-number = depositor.account-number)
    insert into loan values
          (inserted.account-number, inserted.branch-name, – inserted.balance)
   update account set balance = 0
          from account, inserted
          where account.account-number = inserted.account-number
end
```

**Figure 6.5** Example of triggers in MS-SQL server syntax



easier way to maintain summary data. Designers also used triggers extensively for replicating databases; they used triggers on insert/delete/update of each relation to record the changes in relations called **change** or **delta** relations. A separate process copied over the changes to the replica (copy) of the database, and the system executed the changes on the replica. Modern database systems, however, provide built-in facilities for database replication, making triggers unnecessary for replication in most cases.

In fact, many trigger applications, including our example overdraft trigger, can be substituted by "encapsulation" features being introduced in SQL:1999. Encapsulation can be used to ensure that updates to the *balance* attribute of *account* are done only through a special procedure. That procedure would in turn check for negative balance, and carry out the actions of the overdraft trigger. Encapsulations can replace the reorder trigger in a similar manner.

Triggers should be written with great care, since a trigger error detected at run time causes the failure of the insert/delete/update statement that set off the trigger. Furthermore, the action of one trigger can set off another trigger. In the worst case, this could even lead to an infinite chain of triggering. For example, suppose an insert trigger on a relation has an action that causes another (new) insert on the same relation. The insert action then triggers yet another insert action, and so on ad infinitum. Database systems typically limit the length of such chains of triggers (for example to 16 or 32), and consider longer chains of triggering an error.

Triggers are occasionally called *rules*, or *active rules*, but should not be confused with Datalog rules (see Section 5.2), which are really view definitions.

## 6.5 Security and Authorization

The data stored in the database need protection from unauthorized access and malicious destruction or alteration, in addition to the protection against accidental intro-

duction of inconsistency that integrity constraints provide. In this section, we examine the ways in which data may be misused or intentionally made inconsistent. We then present mechanisms to guard against such occurrences.

### 6.5.1 Security Violations

Among the forms of malicious access are:

- Unauthorized reading of data (theft of information)
- Unauthorized modification of data
- Unauthorized destruction of data

**Database security** refers to protection from malicious access. Absolute protection of the database from malicious abuse is not possible, but the cost to the perpetrator can be made high enough to deter most if not all attempts to access the database without proper authority.

To protect the database, we must take security measures at several levels:

- **Database system**. Some database-system users may be authorized to access only a limited portion of the database. Other users may be allowed to issue queries, but may be forbidden to modify the data. It is the responsibility of the database system to ensure that these authorization restrictions are not violated.
- **Operating system**. No matter how secure the database system is, weakness in operating-system security may serve as a means of unauthorized access to the database.
- **Network**. Since almost all database systems allow remote access through terminals or networks, software-level security within the network software is as important as physical security, both on the Internet and in private networks.
- **Physical**. Sites with computer systems must be physically secured against armed or surreptitious entry by intruders.
- **Human**. Users must be authorized carefully to reduce the chance of any user giving access to an intruder in exchange for a bribe or other favors.



Security at all these levels must be maintained if database security is to be ensured. A weakness at a low level of security (physical or human) allows circumvention of strict high-level (database) security measures.

In the remainder of this section, we shall address security at the database-system level. Security at the physical and human levels, although important, is beyond the scope of this text.

Security within the operating system is implemented at several levels, ranging from passwords for access to the system to the isolation of concurrent processes running within the system. The file system also provides some degree of protection. The

The SQL data-definition language includes commands to grant and revoke privileges. The **grant** statement is used to confer authorization. The basic form of this statement is:

**grant** <privilege list> **on** <relation name or view name> **to** <user/role list>

The *privilege list* allows the granting of several privileges in one command.

The following **grant** statement grants users $U_1$, $U_2$, and $U_3$ **select** authorization on the *account* relation:

**grant select on** *account* **to** $U_1$, $U_2$, $U_3$

The **update** authorization may be given either on all attributes of the relation or on only some. If **update** authorization is included in a **grant** statement, the list of attributes on which update authorization is to be granted optionally appears in parentheses immediately after the **update** keyword. If the list of attributes is omitted, the update privilege will be granted on all attributes of the relation.

This **grant** statement gives users $U_1$, $U_2$, and $U_3$ update authorization on the *amount* attribute of the *loan* relation:

**grant update** (*amount*) **on** *loan* **to** $U_1$, $U_2$, $U_3$



The **insert** privilege may also specify a list of attributes; any inserts to the relation must specify only these attributes, and the system either gives each of the remaining attributes default values (if a default is defined for the attribute) or sets them to null.

The SQL **references** privilege is granted on specific attributes in a manner like that for the **update** privilege. The following **grant** statement allows user $U_1$ to create relations that reference the key *branch-name* of the *branch* relation as a foreign key:

**grant references** (*branch-name*) **on** *branch* **to** $U_1$

Initially, it may appear that there is no reason ever to prevent users from creating foreign keys referencing another relation. However, recall from Section 6.2 that foreign-key constraints restrict deletion and update operations on the referenced relation. In the preceding example, if $U_1$ creates a foreign key in a relation $r$ referencing the *branch-name* attribute of the *branch* relation, and then inserts a tuple into $r$ pertaining to the Perryridge branch, it is no longer possible to delete the Perryridge branch from the *branch* relation without also modifying relation $r$. Thus, the definition of a foreign key by $U_1$ restricts future activity by other users; therefore, there is a need for the **references** privilege.

The privilege **all privileges** can be used as a short form for all the allowable privileges. Similarly, the user name **public** refers to all current and future users of the system. SQL also includes a **usage** privilege that authorizes a user to use a specified domain (recall that a domain corresponds to the programming-language notion of a type, and may be user defined).

### 6.6.2 Roles

Roles can be created in SQL:1999 as follows

**create role** *teller*

Roles can then be granted privileges just as the users can, as illustrated in this statement:

**grant select on** *account*
**to** *teller*

Roles can be asigned to the users, as well as to some other roles, as these statements show.

**grant** *teller* **to** john
**create role** *manager*
**grant** *teller* **to** *manager*
**grant** *manager* **to** mary

Thus the privileges of a user or a role consist of

- All privileges directly granted to the user/role
- All privileges granted to roles that have been granted to the user/role



Note that there can be a chain of roles; for example, the role *employee* may be granted to all *teller*s. In turn the role *teller* is granted to all *manager*s. Thus, the *manager* role inherits all privileges granted to the roles *employee* and to *teller* in addition to privileges granted directly to *manager*.

### 6.6.3 The Privilege to Grant Privileges

By default, a user/role that is granted a privilege is not authorized to grant that privilege to another user/role. If we wish to grant a privilege and to allow the recipient to pass the privilege on to other users, we append the **with grant option** clause to the appropriate **grant** command. For example, if we wish to allow $U_1$ the **select** privilege on *branch* and allow $U_1$ to grant this privilege to others, we write

**grant select on** *branch* **to** $U_1$ **with grant option**

To revoke an authorization, we use the **revoke** statement. It takes a form almost identical to that of **grant**:

**revoke** <privilege list> **on** <relation name or view name>
**from** <user/role list> [**restrict** | **cascade**]

Thus, to revoke the privileges that we granted previously, we write

- Implementing authorization through application code, rather than specifying it declaratively in SQL, makes it hard to ensure the absence of loopholes. Because of an oversight, one of the application programs may not check for authorization, allowing unauthorized users access to confidential data. Verifying that all application programs make all required authorization checks involves reading through all the application server code, a formidable task in a large system.

## 6.7 Encryption and Authentication

The various provisions that a database system may make for authorization may still not provide sufficient protection for highly sensitive data. In such cases, data may be stored in **encrypted** form. It is not possible for encrypted data to be read unless the reader knows how to decipher (**decrypt**) them. Encryption also forms the basis of good schemes for authenticating users to a database.



### 6.7.1 Encryption Techniques

There are a vast number of techniques for the encryption of data. Simple encryption techniques may not provide adequate security since it may be easy for an unauthorized user to break the code. As an example of a weak encryption technique, consider the substitution of each character with the next character in the alphabet. Thus,

Perryridge

becomes

Qfsszsjehf

If an unauthorized user sees only "Qfsszsjehf," she probably has insufficient information to break the code. However, if the intruder sees a large number of encrypted branch names, she could use statistical data regarding the relative frequency of characters to guess what substitution is being made (for example, *E* is the most common letter in English text, followed by *T, A, O, N, I* and so on).

A good encryption technique has the following properties:

- It is relatively simple for authorized users to encrypt and decrypt data.
- It depends not on the secrecy of the algorithm, but rather on a parameter of the algorithm called the *encryption key*.
- Its encryption key is extremely difficult for an intruder to determine.

One approach, the *Data Encryption Standard* (DES), issued in 1977, does both a substitution of characters and a rearrangement of their order on the basis of an encryption key. For this scheme to work, the authorized users must be provided with the encryption key via a secure mechanism. This requirement is a major weakness, since the scheme is no more secure than the security of the mechanism by which the encryption key is transmitted. The DES standard was reaffirmed in 1983, 1987,

and again in 1993. However, weakness in DES was recongnized in 1993 as reaching a point where a new standard to be called the **Advanced Encryption Standard** (AES), needed to be selected. In 2000, the **Rijndael algorithm** (named for the inventors V. Rijmen and J. Daemen), was selected to be the AES. The Rijndael algorithm was chosen for its significantly stronger level of security and its relative ease of implementation on current computer systems as well as such devices as smart cards. Like the DES standard, the Rijndael algorithm is a shared-key (or, symmetric key) algorithm in which the authorized users share a key.

**Public-key encryption** is an alternative scheme that avoids some of the problems that we face with the DES. It is based on two keys; a *public key* and a *private key*. Each user $U_i$ has a public key $E_i$ and a private key $D_i$. All public keys are published: They can be seen by anyone. Each private key is known to only the one user to whom the key belongs. If user $U_1$ wants to store encrypted data, $U_1$ encrypts them using its public key $E_1$. Decryption requires the private key $D_1$.

Because the encryption key for each user is public, it is possible to exchange information securely by this scheme. If user $U_1$ wants to share data with $U_2$, $U_1$ encrypts the data using $E_2$, the public key of $U_2$. Since only user $U_2$ knows how to decrypt the data, information is transferred securely.

For public-key encryption to work, there must be a scheme for encryption that can be made public without making it easy for people to figure out the scheme for decryption. In other words, it must be hard to deduce the private key, given the public key. Such a scheme does exist and is based on these conditions:

- There is an efficient algorithm for testing whether or not a number is prime.
- No efficient algorithm is known for finding the prime factors of a number.

For purposes of this scheme, data are treated as a collection of integers. We create a public key by computing the product of two large prime numbers: $P_1$ and $P_2$. The private key consists of the pair $(P_1, P_2)$. The decryption algorithm cannot be used successfully if only the product $P_1P_2$ is known; it needs the individual values $P_1$ and $P_2$. Since all that is published is the product $P_1P_2$, an unauthorized user would need to be able to factor $P_1P_2$ to steal data. By choosing $P_1$ and $P_2$ to be sufficiently large (over 100 digits), we can make the cost of factoring $P_1P_2$ prohibitively high (on the order of years of computation time, on even the fastest computers).

The details of public-key encryption and the mathematical justification of this technique's properties are referenced in the bibliographic notes.

Although public-key encryption by this scheme is secure, it is also computationally expensive. A hybrid scheme used for secure communication is as follows: DES keys are exchanged via a public-key–encryption scheme, and DES encryption is used on the data transmitted subsequently.

## 6.7.2 Authentication

Authentication refers to the task of verifying the identity of a person/software connecting to a database. The simplest form of authentication consists of a secret password which must be presented when a connection is opened to a database.

The domain of all integers would be nonatomic if we considered each integer to be an ordered list of digits.

As a practical illustration of the above point, consider an organization that assigns employees identification numbers of the following form: The first two letters specify the department and the remaining four digits are a unique number within the department for the employee. Examples of such numbers would be $CS0012$ and $EE1127$. Such identification numbers can be divided into smaller units, and are therefore nonatomic. If a relation schema had an attribute whose domain consists of identification numbers encoded as above, the schema would not be in first normal form.

When such identification numbers are used, the department of an employee can be found by writing code that breaks up the structure of an identification number. Doing so requires extra programming, and information gets encoded in the application program rather than in the database. Further problems arise if such identification numbers are used as primary keys: When an employee changes department, the employee's identification number must be changed everywhere it occurs, which can be a difficult task, or the code that interprets the number would give a wrong result.

The use of set valued attributes can lead to designs with redundant storage of data, which in turn can result in inconsistencies. For instance, instead of the relationship between accounts and customers being represented as a separate relation *depositor*, a database designer may be tempted to store a set of *owners* with each account, and a set of *accounts* with each customer. Whenever an account is created, or the set of owners of an account is updated, the update has to be performed at two places; failure to perform both updates can leave the database in an inconsistent state. Keeping only one of these sets would avoid repeated information, but would complicate some queries. Set valued attributes are also more complicated to write queries with, and more complicated to reason about.



In this chapter we consider only atomic domains, and assume that relations are in first normal form. Although we have not mentioned first normal form earlier, when we introduced the relational model in Chapter 3 we stated that attribute values must be atomic.

Some types of nonatomic values can be useful, although they should be used with care. For example, composite valued attributes are often useful, and set valued attributes are also useful in many cases, which is why both are supported in the E-R model. In many domains where entities have a complex structure, forcing a first normal form representation represents an unnecessary burden on the application programmer, who has to write code to convert data into atomic form. There is also a runtime overhead of converting data back and forth from the atomic form. Support for nonatomic values can thus be very useful in such domains. In fact, modern database systems do support many types of nonatomic values, as we will see in Chapters 8 and 9. However, in this chapter we restrict ourselves to relations in first normal form.

## 7.2 Pitfalls in Relational-Database Design

Before we continue our discussion of normal forms, let us look at what can go wrong in a bad database design. Among the undesirable properties that a bad design may have are:

- Repetition of information
- Inability to represent certain information

We shall discuss these problems with the help of a modified database design for our banking example: In contrast to the relation schema used in Chapters 3 to 6, suppose the information concerning loans is kept in one single relation, *lending*, which is defined over the relation schema

$$\textit{Lending-schema} = (\textit{branch-name}, \textit{branch-city}, \textit{assets}, \textit{customer-name}, \textit{loan-number}, \textit{amount})$$

Figure 7.1 shows an instance of the relation *lending* (*Lending-schema*). A tuple $t$ in the *lending* relation has the following intuitive meaning:

- $t$[*assets*] is the asset figure for the branch named $t$[*branch-name*].
- $t$[*branch-city*] is the city in which the branch named $t$[*branch-name*] is located.
- $t$[*loan-number*] is the number assigned to a loan made by the branch named $t$[*branch-name*] to the customer named $t$[*customer-name*].
- $t$[*amount*] is the amount of the loan whose number is $t$[*loan-number*].

Suppose that we wish to add a new loan to our database. Say that the loan is made by the Perryridge branch to Adams in the amount of $1500. Let the *loan-number* be L-31. In our design, we need a tuple with values on all the attributes of *Lending-schema*. Thus, we must repeat the asset and city data for the Perryridge branch, and must add the tuple

(Perryridge, Horseneck, 1700000, Adams, L-31, 1500)

| *branch-name* | *branch-city* | *assets* | *customer-name* | *loan-number* | *amount* |
|---|---|---|---|---|---|
| Downtown | Brooklyn | 9000000 | Jones | L-17 | 1000 |
| Redwood | Palo Alto | 2100000 | Smith | L-23 | 2000 |
| Perryridge | Horseneck | 1700000 | Hayes | L-15 | 1500 |
| Downtown | Brooklyn | 9000000 | Jackson | L-14 | 1500 |
| Mianus | Horseneck | 400000 | Jones | L-93 | 500 |
| Round Hill | Horseneck | 8000000 | Turner | L-11 | 900 |
| Pownal | Bennington | 300000 | Williams | L-29 | 1200 |
| North Town | Rye | 3700000 | Hayes | L-16 | 1300 |
| Downtown | Brooklyn | 9000000 | Johnson | L-18 | 2000 |
| Perryridge | Horseneck | 1700000 | Glenn | L-25 | 2500 |
| Brighton | Brooklyn | 7100000 | Brooks | L-10 | 2200 |

**Figure 7.1** Sample *lending* relation.

| *loan-number* | *branch-name* | *amount* |
|---|---|---|
| L-17 | Downtown | 1000 |
| L-23 | Redwood | 2000 |
| L-15 | Perryridge | 1500 |
| L-14 | Downtown | 1500 |
| L-93 | Mianus | 500 |
| L-11 | Round Hill | 900 |
| L-29 | Pownal | 1200 |
| L-16 | North Town | 1300 |
| L-18 | Downtown | 2000 |
| L-25 | Perryridge | 2500 |
| L-10 | Brighton | 2200 |

**Figure 7.4** The *loan* relation.



can have streets with the same name. Thus it is possible, at some time, to have an instance of the *customer* relation in which *customer-street* → *customer-city* is not satisfied. So, we would not include *customer-street* → *customer-city* in the set of functional dependencies that hold on *Customer-schema*.

In the *loan* relation (on *Loan-schema*) of Figure 7.4, we see that the dependency *loan-number* → *amount* is satisfied. In contrast to the case of *customer-city* and *customer-street* in *Customer-schema*, we do believe that the real-world enterprise that we are modeling requires each loan to have only one amount. Therefore, we want to require that *loan-number* → *amount* be satisfied by the *loan* relation at all times. In other words, we require that the constraint *loan-number* → *amount* hold on *Loan-schema*.

In the *branch* relation of Figure 7.5, we see that *branch-name* → *assets* is satisfied, as is *assets* → *branch-name*. We want to require that *branch-name* → *assets* hold on *Branch-schema*. However, we do not wish to require that *assets* → *branch-name* hold, since it is possible to have several branches that have the same asset value.

In what follows, we assume that, when we design a relational database, we first list those functional dependencies that must always hold. In the banking example, our list of dependencies includes the following:

| *branch-name* | *branch-city* | *assets* |
|---|---|---|
| Downtown | Brooklyn | 9000000 |
| Redwood | Palo Alto | 2100000 |
| Perryridge | Horseneck | 1700000 |
| Mianus | Horseneck | 400000 |
| Round Hill | Horseneck | 8000000 |
| Pownal | Bennington | 300000 |
| North Town | Rye | 3700000 |
| Brighton | Brooklyn | 7100000 |

**Figure 7.5** The *branch* relation.

2. If $B$ is deleted, we get the set $\{A \rightarrow C, B \rightarrow AC$, and $C \rightarrow AB\}$. This case is symmetrical to the previous case, leading to the canonical covers

$$F_c = \{A \rightarrow C, C \rightarrow B, \text{ and } B \rightarrow A\}, \text{ and}$$
$$F_c = \{A \rightarrow C, B \rightarrow C, \text{ and } C \rightarrow AB\}.$$

As an exercise, can you find one more canonical cover for $F$?

## 7.4 Decomposition

The bad design of Section 7.2 suggests that we should *decompose* a relation schema that has many attributes into several schemas with fewer attributes. Careless decomposition, however, may lead to another form of bad design.

Consider an alternative design in which we decompose *Lending-schema* into the following two schemas:

*Branch-customer-schema* = (*branch-name*, *branch-city*, *assets*, *customer-name*)
*Customer-loan-schema* = (*customer-name*, *loan-number*, *amount*)

Using the *lending* relation of Figure 7.1, we construct our new relations *branch-customer* (*Branch-customer-schema*) and *customer-loan* (*Customer-loan-schema*):

$$branch\text{-}customer = \Pi_{branch\text{-}name,\ branch\text{-}city,\ assets,\ customer\text{-}name}\ (lending)$$
$$customer\text{-}loan = \Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}\ (lending)$$

Figures 7.9 and 7.10, respectively, show the resulting *branch-customer* and *customer-name* relations.

Of course, there are cases in which we need to reconstruct the *loan* relation. For example, suppose that we wish to find all branches that have loans with amounts less than $1000. No relation in our alternative database contains these data. We need to reconstruct the *lending* relation. It appears that we can do so by writing

$$branch\text{-}customer \bowtie customer\text{-}loan$$

| *branch-name* | *branch-city* | *assets* | *customer-name* |
|---|---|---|---|
| Downtown | Brooklyn | 9000000 | Jones |
| Redwood | Palo Alto | 2100000 | Smith |
| Perryridge | Horseneck | 1700000 | Hayes |
| Downtown | Brooklyn | 9000000 | Jackson |
| Mianus | Horseneck | 400000 | Jones |
| Round Hill | Horseneck | 8000000 | Turner |
| Pownal | Bennington | 300000 | Williams |
| North Town | Rye | 3700000 | Hayes |
| Downtown | Brooklyn | 9000000 | Johnson |
| Perryridge | Horseneck | 1700000 | Glenn |
| Brighton | Brooklyn | 7100000 | Brooks |

**Figure 7.9** The relation *branch-customer*.

| *branch-name* | *branch-city* | *assets* | *customer-name* | *loan-number* | *amount* |
|---|---|---|---|---|---|
| Downtown | Brooklyn | 9000000 | Jones | L-17 | 1000 |
| Downtown | Brooklyn | 9000000 | Jones | L-93 | 500 |
| Redwood | Palo Alto | 2100000 | Smith | L-23 | 2000 |
| Perryridge | Horseneck | 1700000 | Hayes | L-15 | 1500 |
| Perryridge | Horseneck | 1700000 | Hayes | L-16 | 1300 |
| Downtown | Brooklyn | 9000000 | Jackson | L-14 | 1500 |
| Mianus | Horseneck | 400000 | Jones | L-17 | 1000 |
| Mianus | Horseneck | 400000 | Jones | L-93 | 500 |
| Round Hill | Horseneck | 8000000 | Turner | L-11 | 900 |
| Pownal | Bennington | 300000 | Williams | L-29 | 1200 |
| North Town | Rye | 3700000 | Hayes | L-15 | 1500 |
| North Town | Rye | 3700000 | Hayes | L-16 | 1300 |
| Downtown | Brooklyn | 9000000 | Johnson | L-58 | 2000 |
| Perryridge | Horseneck | 1700000 | Glenn | L-25 | 2500 |
| Brighton | Brooklyn | 7100000 | Brooks | L-10 | 2200 |

**Figure 7.11** The relation *branch-customer* $\bowtie$ *customer-loan*.



tion. It should be clear from our example that a lossy-join decomposition is, in general, a bad database design.

Why is the decomposition lossy? There is one attribute in common between *Branch-customer-schema* and *Customer-loan-schema*:

$$\textit{Branch-customer-schema} \cap \textit{Customer-loan-schema} = \{\textit{customer-name}\}$$

The only way that we can represent a relationship between, for example, *loan-number* and *branch-name* is through *customer-name*. This representation is not adequate because a customer may have several loans, yet these loans are not necessarily obtained from the same branch.

Let us consider another alternative design, in which we decompose *Lending-schema* into the following two schemas:

$$\textit{Branch-schema} = (\textit{branch-name}, \textit{branch-city}, \textit{assets})$$
$$\textit{Loan-info-schema} = (\textit{branch-name}, \textit{customer-name}, \textit{loan-number}, \textit{amount})$$

There is one attribute in common between these two schemas:

$$\textit{Branch-loan-schema} \cap \textit{Customer-loan-schema} = \{\textit{branch-name}\}$$

Thus, the only way that we can represent a relationship between, for example, *customer-name* and *assets* is through *branch-name*. The difference between this example and the preceding one is that the assets of a branch are the same, regardless of the customer to which we are referring, whereas the lending branch associated with a certain loan amount *does* depend on the customer to which we are referring. For a given *branch-name*, there is exactly one *assets* value and exactly one *branch-city*;

Later in this chapter, we shall introduce constraints other than functional dependencies. We say that a relation is **legal** if it satisfies all rules, or constraints, that we impose on our database.

Let $C$ represent a set of constraints on the database, and let $R$ be a relation schema. A decomposition $\{R_1, R_2, \ldots, R_n\}$ of $R$ is a **lossless-join decomposition** if, for all relations $r$ on schema $R$ that are legal under $C$,

$$r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie \cdots \bowtie \Pi_{R_n}(r)$$

We shall show how to test whether a decomposition is a lossless-join decomposition in the next few sections. A major part of this chapter deals with the questions of how to specify constraints on the database, and how to obtain lossless-join decompositions that avoid the pitfalls represented by the examples of bad database designs that we have seen in this section.



## 7.5 Desirable Properties of Decomposition

We can use a given set of functional dependencies in designing a relational database in which most of the undesirable properties discussed in Section 7.2 do not occur. When we design such systems, it may become necessary to decompose a relation into several smaller relations.

In this section, we outline the desirable properties of a decomposition of a relational schema. In later sections, we outline specific ways of decomposing a relational schema to get the properties we desire. We illustrate our concepts with the *Lending-schema* schema of Section 7.2:

$$\textit{Lending-schema} = (\textit{branch-name}, \textit{branch-city}, \textit{assets}, \textit{customer-name}, \textit{loan-number}, \textit{amount})$$

The set $F$ of functional dependencies that we require to hold on *Lending-schema* are

$$\textit{branch-name} \rightarrow \textit{branch-city assets}$$
$$\textit{loan-number} \rightarrow \textit{amount branch-name}$$

As we discussed in Section 7.2, *Lending-schema* is an example of a bad database design. Assume that we decompose it to the following three relations:

$$\textit{Branch-schema} = (\textit{branch-name}, \textit{branch-city}, \textit{assets})$$
$$\textit{Loan-schema} = (\textit{loan-number}, \textit{branch-name}, \textit{amount})$$
$$\textit{Borrower-schema} = (\textit{customer-name}, \textit{loan-number})$$

We claim that this decomposition has several desirable properties, which we discuss next. Note that these three relation schemas are precisely the ones that we used previously, in Chapters 3 through 5.

### 7.5.1 Lossless-Join Decomposition

In Section 7.2, we argued that, when we decompose a relation into a number of smaller relations, it is crucial that the decomposition be lossless. We claim that the

CHAPTER 25

# Oracle

Hakan Jakobsson
Oracle Corporation



When Oracle was founded in 1977 as Software Development Laboratories by Larry Ellison, Bob Miner, and Ed Oates, there were no commercial relational database products. The company, which was later renamed Oracle, set out to build a relational database management system as a commercial product, and was the first to reach the market. Since then, Oracle has held a leading position in the relational database market, but over the years its product and service offerings have grown beyond the relational database server. In addition to tools directly related to database development and management, Oracle sells business intelligence tools, including a multidimensional database management system (Oracle Express), query and analysis tools, data-mining products, and an application server with close integration to the database server.

In addition to database-related servers and tools, the company also offers application software for enterprise resource planning and customer-relationship management, including areas such as financials, human resources, manufacturing, marketing, sales, and supply chain management. Oracle's Business OnLine unit offers services in these areas as an application service provider.

This chapter surveys a subset of the features, options, and functionality of Oracle products. New versions of the products are being developed continually, so all product descriptions are subject to change. The feature set described here is based on the first release of Oracle9*i*.

## 25.1 Database Design and Querying Tools

Oracle provides a variety of tools for database design, querying, report generation and data analysis, including OLAP.

variables into single units. Oracle supports SQLJ (SQL embedded in Java) and JDBC, and provides a tool to generate Java class definitions corresponding to user-defined database types.

### 25.2.2 Triggers

Oracle provides several types of triggers and several options for when and how they are invoked. (See Section 6.4 for an introduction to triggers in SQL.) Triggers can be written in PL/SQL or Java or as C callouts.

For triggers that execute on DML statements such as insert, update, and delete, Oracle supports **row triggers** and **statement triggers**. Row triggers execute once for every row that is affected (updated or deleted, for example) by the DML operation. A statement trigger is executed just once per statement. In each case, the trigger can be defined as either a *before* or *after* trigger, depending on whether it is to be invoked before or after the DML operation is carried out.

Oracle allows the creation of **instead of** triggers for views that cannot be subject to DML operations. Depending on the view definition, it may not be possible for Oracle to translate a DML statement on a view to modifications of the underlying base tables unambiguously. Hence DML operations on views are subject to numerous restrictions. A user can create an **instead of** trigger on a view to specify manually what operations on the base tables are to occur in response to the DML operation on the view. Oracle executes the trigger instead of the DML operation and therefore provides a mechanism to circumvent the restrictions on DML operations against views.

Oracle also has triggers that execute on a variety of other events, like database startup or shutdown, server error messages, user logon or logoff, and DDL statements such as **create**, **alter** and **drop** statements.



## 25.3 Storage and Indexing

In Oracle parlance, a *database* consists of information stored in files and is accessed through an *instance*, which is a shared memory area and a set of processes that interact with the data in the files.

### 25.3.1 Table Spaces

A database consists of one or more logical storage units called **table spaces**. Each table space, in turn, consists of one or more physical structures called **data files**. These may be either files managed by the operating system or raw devices.

Usually, an Oracle database will have the following table spaces:

- The **system** table space, which is always created. It contains the data dictionary tables and storage for triggers and stored procedures.
- Table spaces created to store user data. While user data can be stored in the **system** table space, it is often desirable to separate the user data from the system data. Usually, the decision about what other table spaces should be created is based on performance, availability, maintainability, and ease of admin-

All operations are performed directly on the compressed representation of the bitmaps—no decompression is necessary—and the resulting (compressed) bitmap represents those rows that match all the logical conditions.

The ability to use the Boolean operations to combine multiple indices is not limited to bitmap indices. Oracle can convert row-ids to the compressed bitmap representation, so it can use a regular B-tree index anywhere in a Boolean tree of bitmap operation simply by putting a row-id-to-bitmap operator on top of the index access in the execution plan.

As a rule of thumb, bitmap indices tend to be more space efficient than regular B-tree indices if the number of distinct key values is less than half the number of rows in the table. For example, in a table with 1 million rows, an index on a column with less than 500,000 distinct values would probably be smaller if it were created as a bitmap index. For columns with a very small number of distinct values—for example, columns referring to properties such as country, state, gender, marital status, and various status flags—a bitmap index might require only a small fraction of the space of a regular B-tree index. Any such space advantage can also give rise to corresponding performance advantages in the form of fewer disk I/Os when the index is scanned.



### 25.3.7 Function-Based Indices

In addition to creating indices on one or multiple columns of a table, Oracle allows indices to be created on expressions that involve one or more columns, such as $col_1 + col_2 * 5$. For example, by creating an index on the expression *upper*(*name*), where *upper* is a function that returns the uppercase version of a string, and *name* is a column, it is possible to do case-insensitive searches on the *name* column. In order to find all rows with name "van Gogh" efficiently, the condition

$$upper(name) = \text{'VAN GOGH'}$$

would be used in the **where** clause of the query. Oracle then matches the condition with the index definition and concludes that the index can be used to retrieve all the rows matching "van Gogh" regardless of how the name was capitalized when it was stored in the database. A function-based index can be created as either a bitmap or a B-tree index.

### 25.3.8 Join Indices

A join index is an index where the key columns are not in the table that is referenced by the row-ids in the index. Oracle supports bitmap join indices primarily for use with star schemas (see Section 22.4.2). For example, if there is a column for product names in a product dimension table, a bitmap join index on the fact table with this key column could be used to retrieve the fact table rows that correspond to a product with a specific name, although the name is not stored in the fact table. How the rows in the fact and dimension tables correspond is based on a join condition that is specified when the index is created, and becomes part of the index metadata. When a query is

the correct result. For example, if a query needs sales by quarter, the rewrite can take advantage of a view that materializes sales by month, by adding additional aggregation to roll up the months to quarters. Oracle has a type of metadata object called dimension that allows hierarchical relationships in tables to be defined. For example, for a time dimension table in a star schema, Oracle can define a dimension metadata object to specify how days roll up to months, months to quarters, quarters to years, and so forth. Likewise, hierarchical properties relating to geography can be specified —for example, how sales districts roll up to regions. The query rewrite logic looks at these relationships since they allow a materialized view to be used for wider classes of queries.

The container object for a materialized view is a table, which means that a materialized view can be indexed, partitioned, or subjected to other controls, to improve query performance.

When there are changes to the data in the tables referenced in the query that defines a materialized view, the materialized view must be refreshed to reflect those changes. Oracle supports both full refresh of a materialized view and fast, incremental refresh. In a full refresh, Oracle recomputes the materialized view from scratch, which may be the best option if the underlying tables have had significant changes, for example, changes due to a bulk load. In an incremental refresh, Oracle updates the view using the records that were changed in the underlying tables; the refresh to the view is immediate, that is, it is executed as part of the transaction that changed the underlying tables. Incremental refresh may be better if the number of rows that were changed is low. There are some restrictions on the classes of queries for which a materialized view can be incrementally refreshed (and others for when a materialized view can be created at all).



A materialized view is similar to an index in the sense that, while it can improve query performance, it uses up space, and creating and maintaining it consumes resources. To help resolve this tradeoff, Oracle provides a package that can advise a user of the most cost-effective materialized views, given a particular query workload as input.

## 25.4 Query Processing and Optimization

Oracle supports a large variety of processing techniques in its query processing engine. Some of the more important ones are described here briefly.

### 25.4.1 Execution Methods

Data can be accessed through a variety of access methods:

- **Full table scan.** The query processor scans the entire table by getting information about the blocks that make up the table from the extent map, and scanning those blocks.
- **Index scan.** The processor creates a start and/or stop key from conditions in the query and uses it to scan to a relevant part of the index. If there are columns that need to be retrieved, that are not part of the index, the index

one that has been subjected to advanced transformations. Not all query transformation techniques are guaranteed to be beneficial for every query, but by generating a cost estimate for the best plan with and without the transformation applied, Oracle is able to make an intelligent decision.

Some of the major types of transformations and rewrites supported by Oracle are as follows:

- **View merging.** A view reference in a query is replaced by the view definition. This transformation is not applicable to all views.
- **Complex view merging.** Oracle offers this feature for certain classes of views that are not subject to regular view merging because they have a **group by** or **select distinct** in the view definition. If such a view is joined to other tables, Oracle can commute the joins and the sort operation used for the **group by** or **distinct**.
- **Subquery flattening.** Oracle has a variety of transformations that convert various classes of subqueries into joins, semijoins, or antijoins.
- **Materialized view rewrite.** Oracle has the ability to rewrite a query automatically to take advantage of materialized views. If some part of the query can be matched up with an existing materialized view, Oracle can replace that part of the query with a reference to the table in which the view is materialized. If needed, Oracle adds join conditions or **group by** operations to preserve the semantics of the query. If multiple materialized views are applicable, Oracle picks the one that gives the greatest advantage in reducing the amount of data that has to be processed. In addition, Oracle subjects both the rewritten query and the original version to the full optimization process producing an execution plan and an associated cost estimate for each. Oracle then decides whether to execute the rewritten or the original version of the query on the basis of the cost estimates.
- **Star transformation.** Oracle supports a technique for evaluating queries against star schemas, known as the star transformation. When a query contains a join of a fact table with dimension tables, and selections on attributes from the dimension tables, the query is transformed by deleting the join condition between the fact table and the dimension tables, and replacing the selection condition on each dimension table by a subquery of the form:

  *fact_table.*$fk_i$ **in**
  (**select** *pk* **from** *dimension_table*$_i$
  **where** <conditions on *dimension_table*$_i$ >)

  One such subquery is generated for each dimension that has some constraining predicate. If the dimension has a snow-flake schema (see Section 22.4), the subquery will contain a join of the applicable tables that make up the dimension.

Oracle uses the values that are returned from each subquery to probe an index on the corresponding fact table column, getting a bitmap as a result. The bitmaps generated from different subqueries are combined by a bitmap **and** operation. The resultant bitmap can be used to access matching fact table rows. Hence, only those rows in the fact table that simultaneously match the conditions on the constrained dimensions will be accessed.

Both the decision on whether the use of a subquery for a particular dimension is cost-effective, and the decision on whether the rewritten query is better than the original, are based on the optimizer's cost estimates.

### 25.4.2.2 Access Path Selection

Oracle has a cost-based optimizer that determines join order, join methods, and access paths. Each operation that the optimizer considers has an associated cost function, and the optimizer tries to generate the combination of operations that has the lowest overall cost.

In estimating the cost of an operation, the optimizer relies on statistics that have been computed for schema objects such as tables and indexes. The statistics contain information about the size of the object, the cardinality, data distribution of table columns, and so forth. For column statistics, Oracle supports height-balanced and frequency histograms. To facilitate the collection of optimizer statistics, Oracle can monitor modification activity on tables and keep track of those tables that have been subject to enough changes that recalculating the statistics may be appropriate. Oracle also tracks what columns are used in **where** clauses of queries, which make them potential candidates for histogram creation. With a single command, a user can tell Oracle to refresh the statistics for those tables that were marked as sufficiently changed. Oracle uses sampling to speed up the process of gathering the new statistics and automatically chooses the smallest adequate sample percentage. It also determines whether the distribution of the marked columns merit the creation of histograms; if the distribution is close to uniform, Oracle uses a simpler representation of the column statistics.

Oracle uses both CPU cost and disk I/Os in the optimizer cost model. To balance the two components, it stores measures about CPU speed and disk I/O performance as part of the optimizer statistics. Oracle's package for gathering optimizer statistics computes these measures.

For queries involving a nontrivial number of joins, the search space is an issue for a query optimizer. Oracle addresses this issue in several ways. The optimizer generates an initial join order and then decides on the best join methods and access paths for that join order. It then changes the order of the tables and determines the best join methods and access paths for the new join order and so forth, while keeping the best plan that has been found so far. Oracle cuts the optimization short if the number of different join orders that have been considered becomes so large that the time spent in the optimizer may be noticeable compared to the time it would take to execute the best plan found so far. Since this cutoff depends on the cost estimate for the best plan found so far, finding a good plan early is important so that the optimization can be stopped after a smaller number of join orders, resulting in better response time.

tion. This area contains stack space for various session data and the private memory for the SQL statement that it is executing. It also contains memory for sorting and hashing operations that may occur during the evaluation of the statement.

The SGA is a memory area for structures that are shared among users. It is made up by several major structures, including:

- **The buffer cache.** This cache keeps frequently accessed data blocks (from tables as well as indices) in memory to reduce the need to perform physical disk I/O. A least recently used replacement policy is used except for blocks accessed during a full table scan. However, Oracle allows multiple buffer pools to be created that have different criteria for aging out data. Some Oracle operations bypass the buffer cache and read data directly from disk.
- **The redo log buffer.** This buffer contains the part of the redo log that has not yet been written to disk.
- **The shared pool.** Oracle seeks to maximize the number of users that can use the database concurrently by minimizing the amount of memory that is needed for each user. One important concern in this context is the ability to share the internal representation of SQL statements and procedural code written in PL/SQL. When multiple users execute the same SQL statement, they can share most data structures that represent the execution plan for the statement. Only data that is local to each specific invocation of the statement needs to be kept in private memory.

  The sharable parts of the data structures representing the SQL statement are stored in the shared pool, including the text of the statement. The caching of SQL statements in the shared pool also saves compilation time, since a new invocation of a statement that is already cached does not have to go through the complete compilation process. The determination of whether an SQL statement is the same as one existing in the shared pool is based on exact text matching and the setting of certain session parameters. Oracle can automatically replace constants in an SQL statement with bind variables; future queries that are the same except for the values of constants will then match the earlier query in the shared pool. The shared pool also contains caches for dictionary information and various control structures.



### 25.6.2 Dedicated Server: Process Structures

There are two types of processes that execute Oracle server code: server processes that process SQL statements and background processes that perform various administrative and performance-related tasks. Some of these processes are optional, and in some cases, multiple processes of the same type can be used for performance reasons. Some of the most important types of background processes are:

- **Database writer.** When a buffer is removed from the buffer cache, it must be written back to disk if it has been modified since it entered the cache. This task

| *title* | *author* | *pub-name* | *pub-branch* | *keyword* |
|---|---|---|---|---|
| Compilers | Smith | McGraw-Hill | New York | parsing |
| Compilers | Jones | McGraw-Hill | New York | parsing |
| Compilers | Smith | McGraw-Hill | New York | analysis |
| Compilers | Jones | McGraw-Hill | New York | analysis |
| Networks | Jones | Oxford | London | Internet |
| Networks | Frick | Oxford | London | Internet |
| Networks | Jones | Oxford | London | Web |
| Networks | Frick | Oxford | London | Web |

**Figure 9.2** *flat-books*, a 1NF version of non-1NF relation *books*.

Much of the awkwardness of the *flat-books* relation in Figure 9.2 disappears if we assume that the following multivalued dependencies hold:

- *title* $\twoheadrightarrow$ *author*
- *title* $\twoheadrightarrow$ *keyword*
- *title* $\rightarrow$ *pub-name*, *pub-branch*

Then, we can decompose the relation into 4NF using the schemas:

- *authors*(*title*, *author*)
- *keywords*(*title*, *keyword*)
- *books4*(*title*, *pub-name*, *pub-branch*)



Figure 9.3 shows the projection of the relation *flat-books* of Figure 9.2 onto the preceding decomposition.

Although our example book database can be adequately expressed without using nested relations, the use of nested relations leads to an easier-to-understand model: The typical user of an information-retrieval system thinks of the database in terms of books having sets of authors, as the non-1NF design models. The 4NF design would require users to include joins in their queries, thereby complicating interaction with the system.

We could define a non-nested relational view (whose contents are identical to *flat-books*) that eliminates the need for users to write joins in their query. In such a view, however, we lose the one-to-one correspondence between tuples and books.

## 9.2 Complex Types

Nested relations are just one example of extensions to the basic relational model; other nonatomic data types, such as nested records, have also proved useful. The object-oriented data model has caused a need for features such as inheritance and references to objects. With complex type systems and object orientation, we can represent E-R model concepts, such as identity of entities, multivalued attributes, and generalization and specialization directly, without a complex translation to the relational model.

Here, *author-array* is an array of up to 10 author names. We can access elements of an array by specifying the array index, for example *author-array*[1].

Arrays are the only collection type supported by SQL:1999; the syntax used is as in the preceding declaration. SQL:1999 does *not* support unordered sets or multisets, although they may appear in future versions of SQL.[1]

Many current-generation database applications need to store attributes that can be large (of the order of many kilobytes), such as a photograph of a person, or very large (of the order of many megabytes or even gigabytes), such as a high-resolution medical image or video clip. SQL:1999 therefore provides new large-object data types for character data (**clob**) and binary data (**blob**). The letters "lob" in these data types stand for "Large OBject". For example, we may declare attributes

*book-review* **clob**(10KB)
*image* **blob**(10MB)
*movie* **blob**(2GB))

Large objects are typically used in external applications, and it makes little sense to retrieve them in their entirety by SQL. Instead, an application would usually retrieve a "locator" for a large object and then use the locator to manipulate the object from the host language. For instance, JDBC permits the programmer to fetch a large object in smaller pieces, rather than all at once, much like fetching data from an operating system file.



### 9.2.2 Structured Types

Structured types can be declared and used in SQL:1999 as in the following example:

```
create type Publisher as
      (name varchar(20),
       branch varchar(20))
create type Book as
      (title varchar(20),
       author-array varchar(20) array [10],
       pub-date date,
       publisher Publisher,
       keyword-set setof(varchar(20)))
create table books of Book
```

The first statement defines a type called *Publisher*, which has two components: a name and a branch. The second statement defines a structured type *Book*, which contains a *title*, an *author-array*, which is an array of authors, a publication date, a publisher (of type *Publisher*), and a set of keywords. (The declaration of *keyword-set* as a set uses our extended syntax, and is not supported by the SQL:1999 standard.) The types illustrated above are called **structured type**s in SQL:1999.

1. The Oracle 8 database system supports nested relations, but uses a syntax different from that in this chapter.

information without using inheritance. We would have to add appropriate referential integrity constraints to ensure that students and teachers are also represented in the *people* table.

## 9.4 Reference Types

Object-oriented languages provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specified type. For example, in SQL:1999 we can define a type *Department* with a field *name* and a field *head* which is a reference to the type *Person*, and a table *departments* of type *Department*, as follows:

```
create type Department (
        name varchar(20),
        head ref(Person) scope people
)
create table departments of Department
```



Here, the reference is restricted to tuples of the table *people*. The restriction of the **scope** of a reference to tuples of a table is mandatory in SQL:1999, and it makes references behave like foreign keys.

We can omit the declaration **scope** *people* from the type declaration and instead make an addition to the **create table** statement:

```
create table departments of Department
        (head with options scope people)
```

In order to initialize a reference attribute, we need to get the identifier of the tuple that is to be referenced. We can get the identifier value of a tuple by means of a query. Thus, to create a tuple with the reference value, we may first create the tuple with a null reference and then set the reference separately:

```
insert into departments
        values ('CS', null)
update departments
        set head = (select ref(p)
                    from people as p
                    where name = 'John')
where name = 'CS'
```

This syntax for accessing the identifier of a tuple is based on the Oracle syntax. SQL:1999 adopts a different approach, one where the referenced table must have an attribute that stores the identifier of the tuple. We declare this attribute, called the **self-referential attribute**, by adding a **ref is** clause to the **create table** statement:

```
create table people of Person
ref is oid system generated
```

If we know that a particular book has three authors, we could write:

**select** *author-array*[1], *author-array*[2], *author-array*[3]
**from** *books*
**where** *title* = 'Database System Concepts'

Now, suppose that we want a relation containing pairs of the form "title, author-name" for each book and each author of the book. We can use this query:

**select** *B.title*, *A.name*
**from** *books* **as** *B*, **unnest**(*B.author-array*) **as** *A*

Since the *author-array* attribute of *books* is a collection-valued field, it can be used in a **from** clause, where a relation is expected.



### 9.5.3 Nesting and Unnesting

The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called **unnesting**. The *books* relation has two attributes, *author-array* and *keyword-set*, that are collections, and two attributes, *title* and *publisher*, that are not. Suppose that we want to convert the relation into a single flat relation, with no nested relations or structured types as attributes. We can use the following query to carry out the task:

**select** *title*, *A* **as** *author*, *publisher.name* **as** *pub-name*, *publisher.branch*
**as** *pub-branch*, *K* **as** *keyword*
**from** *books* **as** *B*, **unnest**(*B.author-array*) **as** *A*, **unnest** (*B.keyword-set*) **as** *K*

The variable *B* in the from clause is declared to range over *books*. The variable *A* is declared to range over the authors in *author-array* for the book *B*, and *K* is declared to range over the keywords in the *keyword-set* of the book *B*. Figure 9.1 (in Section 9.1) shows an instance *books* relation, and Figure 9.2 shows the 1NF relation that is the result of the preceding query.

The reverse process of transforming a 1NF relation into a nested relation is called **nesting**. Nesting can be carried out by an extension of grouping in SQL. In the normal use of grouping in SQL, a temporary multiset relation is (logically) created for each group, and an aggregate function is applied on the temporary relation. By returning the multiset instead of applying the aggregate function, we can create a nested relation. Suppose that we are given a 1NF relation *flat-books*, as in Figure 9.2. The following query nests the relation on the attribute *keyword*:

**select** *title*, *author*, Publisher(*pub-name*, *pub-branch*) **as** *publisher*,
**set**(*keyword*) **as** *keyword-set*
**from** *flat-books*
**groupby** *title*, *author*, *publisher*

The result of the query on the *books* relation from Figure 9.2 appears in Figure 9.4. If we want to nest the author attribute as well, and thereby to convert the 1NF table

Many object-relational database systems are built on top of existing relational database systems. To do so, the complex data types supported by object-relational systems need to be translated to the simpler type system of relational databases.

To understand how the translation is done, we need only look at how some features of the E-R model are translated into relations. For instance, multivalued attributes in the E-R model correspond to set-valued attributes in the object-relational model. Composite attributes roughly correspond to structured types. ISA hierarchies in the E-R model correspond to table inheritance in the object-relational model. The techniques for converting E-R model features to tables, which we saw in Section 2.9, can be used, with some extensions, to translate object-relational data to relational data.

## 9.8 Summary

- The object-relational data model extends the relational data model by providing a richer type system including collection types, and object orientation.
- Object orientation provides inheritance with subtypes and subtables, as well as object (tuple) references.
- Collection types include nested relations, sets, multisets, and arrays, and the object-relational model permits attributes of a table to be collections.
- The SQL:1999 standard extends the SQL data definition and query language to deal with the new data types and with object orientation.
- We saw a variety of features of the extended data-definition language, as well as the query language, and in particular support for collection-valued attributes, inheritance, and tuple references. Such extensions attempt to preserve the relational foundations—in particular, the declarative access to data—while extending the modeling power.
- Object-relational database systems (that is, database systems based on the object-relation model) provide a convenient migration path for users of relational databases who wish to use object-oriented features.
- We have also outlined the procedural extensions provided by SQL:1999.
- We discussed differences between persistent programming languages and object-relational systems, and mention criteria for choosing between them.



## Review Terms

- Nested relations
- Nested relational model
- Complex types
- Collection types
- Large object types
- Sets
- Arrays
- Multisets
- Character large object (clob)
- Binary large object (blob)

- Structured types
- Methods
- Row types
- Constructors
- Inheritance
  - Single inheritance
  - Multiple inheritance
- Type inheritance
- Most-specific type
- Table inheritance
- Subtable
- Overlapping subtables
- Reference types
- Scope of a reference
- Self-referential attribute
- Path expressions
- Nesting and unnesting
- SQL functions and procedures
- Procedural constructs
- Exceptions
- Handlers
- External language routines



## Exercises

**9.1** Consider the database schema:

*Emp* = (*ename*, **setof**(*Children*), **setof**(*Skills*))
*Children* = (*name*, *Birthday*)
*Birthday* = (*day*, *month*, *year*)
*Skills* = (*type*, **setof**(*Exams*))
*Exams* = (*year*, *city*)

Assume that attributes of type **setof**(*Children*), **setof**(*Skills*), and **setof**(*Exams*), have attribute names *ChildrenSet*, *SkillsSet*, and *ExamsSet*, respectively. Suppose that the database contains a relation *emp (Emp)*. Write the following queries in SQL:1999 (with the extensions described in this chapter).

**a.** Find the names of all employees who have a child who has a birthday in March.

**b.** Find those employees who took an examination for the skill type "typing" in the city "Dayton".

**c.** List all skill types in the relation *emp*.

**9.2** Redesign the database of Exercise 9.1 into first normal form and fourth normal form. List any functional or multivalued dependencies that you assume. Also list all referential-integrity constraints that should be present in the first- and fourth-normal-form schemas.

**9.3** Consider the schemas for the table *people*, and the tables *students* and *teachers*, which were created under *people*, in Section 9.3. Give a relational schema in third normal form that represents the same information. Recall the constraints on subtables, and give all constraints that must be imposed on the relational schema so that every database instance of the relational schema can also be represented by an instance of the schema with inheritance.

```
...
    <account acct-type= "checking">
        <account-number> A-102 </account-number>
        <branch-name> Perryridge </branch-name>
        <balance> 400 </balance>
    </account>
...
```

**Figure 10.4** Use of attributes.

it created would not duplicate tags used by any business partner's XML documents, it can prepend a unique identifier with a colon to each tag name. The bank may use a Web URL such as

http://www.FirstBank.com

as a unique identifier. Using long unique identifiers in every tag would be rather inconvenient, so the namespace standard provides a way to define an abbreviation for identifiers.

In Figure 10.5, the root element (bank) has an attribute xmlns:FB, which declares that FB is defined as an abbreviation for the URL given above. The abbreviation can then be used in various element tags, as illustrated in the figure.

A document can have more than one namespace, declared as part of the root element. Different elements can then be associated with different namespaces. A *default namespace* can be defined, by using the attribute xmlns instead of xmlns:FB in the root element. Elements without an explicit namespace prefix would then belong to the default namespace.

Sometimes we need to store values containing tags without having the tags interpreted as XML tags. So that we can do so, XML allows this construct:

<![CDATA[<account> ···</account>]]>

Because it is enclosed within CDATA, the text <account> is treated as normal text data, not as a tag. The term CDATA stands for character data.

```
<bank xmlns:FB="http://www.FirstBank.com">
    ...
    <FB:branch>
        <FB:branchname> Downtown </FB:branchname>
        <FB:branchcity> Brooklyn </FB:branchcity>
    </FB:branch>
    ...
</bank>
```

**Figure 10.5** Unique tag names through the use of namespaces.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="bank" type="BankType" />
<xsd:element name="account">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="account-number" type="xsd:string"/>
            <xsd:element name="branch-name" type="xsd:string"/>
            <xsd:element name="balance" type="xsd:decimal"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="customer">
            <xsd:element name="customer-number" type="xsd:string"/>
            <xsd:element name="customer-street" type="xsd:string"/>
            <xsd:element name="customer-city" type="xsd:string"/>
</xsd:element>
<xsd:element name="depositor">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="customer-name" type="xsd:string"/>
            <xsd:element name="account-number" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:complexType name="BankType">
    <xsd:sequence>
        <xsd:element ref="account" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="customer" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="depositor" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```



**Figure 10.9** XMLSchema version of DTD from Figure 10.6.

- It allows types to be restricted to create specialized types, for instance by specifying minimum and maximum values.
- It allows complex types to be extended by using a form of inheritance.
- It is a superset of DTDs.
- It allows uniqueness and foreign key constraints.
- It is integrated with namespaces to allow different parts of a document to conform to different schema.
- It is itself specified by XML syntax, as Figure 10.9 shows.

However, the price paid for these features is that XMLSchema is significantly more complicated than DTDs.

## 10.4 Querying and Transformation

Given the increasing number of applications that use XML to exchange, mediate, and store data, tools for effective management of XML data are becoming increasingly important. In particular, tools for querying and transformation of XML data are essential to extract information from large bodies of XML data, and to convert data between different representations (schemas) in XML. Just as the output of a relational query is a relation, the output of an XML query can be an XML document. As a result, querying and transformation can be combined into a single tool.

Several languages provide increasing degrees of querying and transformation capabilities:

- XPath is a language for path expressions, and is actually a building block for the remaining two query languages.
- XSLT was designed to be a transformation language, as part of the XSL style sheet system, which is used to control the formatting of XML data into HTML or other formatting or display languages. Although designed for formatting, XSLT can generate XML as output, and can express many interesting queries. Furthermore, it is currently the most widely available language for manipulating XML data.
- XQuery has been proposed as a standard for querying of XML data. XQuery combines features from many of the earlier proposals for querying XML, in particular the language Quilt.



A **tree model** of XML data is used in all these languages. An XML document is modeled as a **tree**, with **nodes** corresponding to elements and attributes. Element nodes can have children nodes, which can be subelements or attributes of the element. Correspondingly, each node (whether attribute or element), other than the root element, has a parent node, which is an element. The order of elements and attributes in the XML document is modeled by the ordering of children of nodes of the tree. The terms parent, child, ancestor, descendant, and siblings are interpreted in the tree model of XML data.

The text content of an element can be modeled as a text node child of the element. Elements containing text broken up by intervening subelements can have multiple text node children. For instance, an element containing "this is a <bold> wonderful </bold> book" would have a subelement child corresponding to the element bold and two text node children corresponding to "this is a" and "book". Since such structures are not commonly used in database data, we shall assume that elements do not contain both text and subelements.

not change substantially. The XQuery language derives from an XML query language called Quilt; most of the XQuery features we outline here are part of Quilt. Quilt itself includes features from earlier languages such as XPath, discussed in Section 10.4.1, and two other XML query languages, XQL and XML-QL.

Unlike XSLT, XQuery does not represent queries in XML. Instead, they appear more like SQL queries, and are organized into "FLWR" (pronounced "flower") expressions comprising four sections: **for**, **let**, **where**, and **return**. The **for** section gives a series of variables that range over the results of XPath expressions. When more than one variable is specified, the results include the Cartesian product of the possible values the variables can take, making the **for** clause similar in spirit to the **from** clause of an SQL query. The **let** clause simply allows complicated expressions to be assigned to variable names for simplicity of representation. The **where** section, like the SQL **where** clause, performs additional tests on the joined tuples from the **for** section. Finally, the **return** section allows the construction of results in XML.

A simple FLWR expression that returns the account numbers for checking accounts is based on the XML document of Figure 10.8, which uses ID and IDREFS:

```
for $x in /bank-2/account
let $acctno := $x/@account-number
where $x/balance > 400
return <account-number> $acctno </account-number>
```



Since this query is simple, the **let** clause is not essential, and the variable $acctno in the **return** clause could be replaced with $x/@account-number. Note further that, since the **for** clause uses XPath expressions, selections may occur within the XPath expression. Thus, an equivalent query may have only **for** and **return** clauses:

```
for $x in /bank-2/account[balance > 400]
return <account-number> $x/@account-number </account-number>
```

However, the **let** clause simplifies complex queries.

Path expressions in XQuery may return a multiset, with repeated nodes. The function distinct applied on a multiset, returns a set without duplication. The distinct function can be used even within a **for** clause. XQuery also provides aggregate functions such as sum and count that can be applied on collections such as sets and multisets. While XQuery does not provide a **group by** construct, aggregate queries can be written by using nested FLWR constructs in place of grouping; we leave details as an exercise for you. Note also that variables assigned by **let** clauses may be set- or multiset-valued, if the path expression on the right-hand side returns a set or multiset value.

Joins are specified in XQuery much as they are in SQL. The join of depositor, account and customer elements in Figure 10.1, which we wrote in XSLT in Section 10.4.2, can be written in XQuery this way:

storage capacity. Today, almost all active data are stored on disks, except in rare cases where they are stored on tape or in optical jukeboxes.

The fastest storage media—for example, cache and main memory—are referred to as **primary storage**. The media in the next level in the hierarchy—for example, magnetic disks—are referred to as **secondary storage**, or **online storage**. The media in the lowest level in the hierarchy—for example, magnetic tape and optical-disk jukeboxes—are referred to as **tertiary storage**, or **offline storage**.

In addition to the speed and cost of the various storage systems, there is also the issue of storage volatility. **Volatile storage** loses its contents when the power to the device is removed. In the hierarchy shown in Figure 11.1, the storage systems from main memory up are volatile, whereas the storage systems below main memory are nonvolatile. In the absence of expensive battery and generator backup systems, data must be written to **nonvolatile storage** for safekeeping. We shall return to this subject in Chapter 17.



## 11.2 Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Disk capacities have been growing at over 50 percent per year, but the storage requirements of large applications have also been growing very fast, in some cases even faster than the growth rate of disk capacities. A large database may require hundreds of disks.

### 11.2.1 Physical Characteristics of Disks

Physically, disks are relatively simple (Figure 11.2). Each disk **platter** has a flat circular shape. Its two surfaces are covered with a magnetic material, and information is recorded on the surfaces. Platters are made from rigid metal or glass and are covered (usually on both sides) with magnetic recording material. We call such magnetic disks **hard disks**, to distinguish them from **floppy disks**, which are made from flexible material.

When the disk is in use, a drive motor spins it at a constant high speed (usually 60, 90, or 120 revolutions per second, but disks running at 250 revolutions per second are available). There is a read–write head positioned just above the surface of the platter. The disk surface is logically divided into **tracks**, which are subdivided into **sectors**. A **sector** is the smallest unit of information that can be read from or written to the disk. In currently available disks, sector sizes are typically 512 bytes; there are over 16,000 tracks on each platter, and 2 to 4 platters per disk. The inner tracks (closer to the spindle) are of smaller length, and in current-generation disks, the outer tracks contain more sectors than the inner tracks; typical numbers are around 200 sectors per track in the inner tracks, and around 400 sectors per track in the outer tracks. The numbers above vary among different models; higher-capacity models usually have more sectors per track and more tracks on each platter.

The **read–write head** stores information on a sector magnetically as reversals of the direction of magnetization of the magnetic material. There may be hundreds of concentric tracks on a disk surface, containing thousands of sectors.

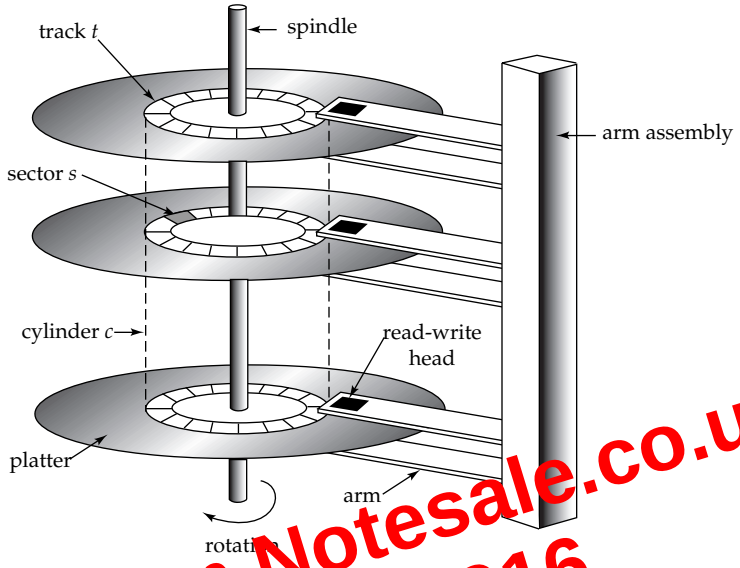



**Figure 11.2** Moving-head disk mechanism.



Each side of a platter of a disk has a read–write head, which moves across the platter to access different tracks. A disk typically contains many platters, and the read–write heads of all the tracks are mounted on a single assembly called a **disk arm**, and move together. The disk platters mounted on a spindle and the heads mounted on a disk arm are together known as **head–disk assemblies**. Since the heads on all the platters move together, when the head on one platter is on the $i$th track, the heads on all other platters are also on the $i$th track of their respective platters. Hence, the $i$th tracks of all the platters together are called the $i$th **cylinder**.

Today, disks with a platter diameter of $3\frac{1}{2}$ inches dominate the market. They have a lower cost and faster seek times (due to smaller seek distances) than do the larger-diameter disks (up to 14 inches) that were common earlier, yet they provide high storage capacity. Smaller-diameter disks are used in portable devices such as laptop computers.

The read–write heads are kept as close as possible to the disk surface to increase the recording density. The head typically floats or flies only microns from the disk surface; the spinning of the disk creates a small breeze, and the head assembly is shaped so that the breeze keeps the head floating just above the disk surface. Because the head floats so close to the surface, platters must be machined carefully to be flat. Head crashes can be a problem. If the head contacts the disk surface, the head can scrape the recording medium off the disk, destroying the data that had been there. Usually, the head touching the surface causes the removed medium to become airborne and to come between the other heads and their platters, causing more crashes. Under normal circumstances, a head crash results in failure of the entire disk, which must then be replaced. Current-generation disk drives use a thin film of magnetic

disks to personal computers and workstations. Mainframe and server systems usually have a faster and more expensive interface, such as high-capacity versions of the SCSI interface, and the Fibre Channel interface.

While disks are usually connected directly by cables to the disk controller, they can be situated remotely and connected by a high-speed network to the disk controller. In the **storage area network** (**SAN**) architecture, large numbers of disks are connected by a high-speed network to a number of server computers. The disks are usually organized locally using **redundant arrays of independent disks** (**RAID**) storage organizations, but the RAID organization may be hidden from the server computers: the disk subsystems pretend each RAID system is a very large and very reliable disk. The controller and the disk continue to use SCSI or Fibre Channel interfaces to talk with each other, although they may be separated by a network. Remote access to disks across a storage area network means that disks can be shared by multiple computers, which could run different parts of an application in parallel. Remote access also means that disks containing important data can be kept in a central server room where they can be monitored and maintained by system administrators, instead of being scattered in different parts of an organization.



### 11.2.2 Performance Measures of Disks

The main measures of the qualities of a disk are capacity, access time, data-transfer rate, and reliability.

**Access time** is the time from when a read or write request is issued to when data transfer begins. To access (that is, to read or write) data on a given sector of a disk, the arm first must move so that it is positioned over the correct track, and then must wait for the sector to appear under it as the disk rotates. The time for repositioning the arm is called the **seek time**, and it increases with the distance that the arm must move. Typical seek times range from 2 to 30 milliseconds, depending on how far the track is from the initial arm position. Smaller disks tend to have lower seek times since the head has to travel a smaller distance.

The **average seek time** is the average of the seek times, measured over a sequence of (uniformly distributed) random requests. If all tracks have the same number of sectors, and we disregard the time required for the head to start moving and to stop moving, we can show that the average seek time is one-third the worst case seek time. Taking these factors into account, the average seek time is around one-half of the maximum seek time. Average seek times currently range between 4 milliseconds and 10 milliseconds, depending on the disk model.

Once the seek has started, the time spent waiting for the sector to be accessed to appear under the head is called the **rotational latency time**. Rotational speeds of disks today range from 5400 rotations per minute (90 rotations per second) up to 15,000 rotations per minute (250 rotations per second), or, equivalently, 4 milliseconds to 11.1 milliseconds per rotation. On an average, one-half of a rotation of the disk is required for the beginning of the desired sector to appear under the head. Thus, the **average latency time** of the disk is one-half the time for a full rotation of the disk.

The access time is then the sum of the seek time and the latency, and ranges from 8 to 20 milliseconds. Once the first sector of the data to be accessed has come under

the head, data transfer begins. The **data-transfer rate** is the rate at which data can be retrieved from or stored to the disk. Current disk systems claim to support maximum transfer rates of about 25 to 40 megabytes per second, although actual transfer rates may be significantly less, at about 4 to 8 megabytes per second.

The final commonly used measure of a disk is the **mean time to failure (MTTF)**, which is a measure of the reliability of the disk. The mean time to failure of a disk (or of any other system) is the amount of time that, on average, we can expect the system to run continuously without any failure. According to vendors' claims, the mean time to failure of disks today ranges from 30,000 to 1,200,000 hours—about 3.4 to 136 years. In practice the claimed mean time to failure is computed on the probability of failure when the disk is new—the figure means that given 1000 relatively new disks, if the MTTF is 1,200,000 hours, on an average one of them will fail in 1200 hours. A mean time to failure of 1,200,000 hours does not imply that the disk can be expected to function for 136 years! Most disks have an expected life span of about 5 years, and have significantly higher rates of failure once they become more than a few years old.

There may be multiple disks sharing a disk interface. The widely used ATA-4 interface standard (also called Ultra DMA) supports 33 megabytes per second transfer rates, while ATA-5 supports 66 megabytes per second. SCSI-3 (Ultra2 wide SCSI) supports 40 megabytes per second, while the more expensive Fibre Channel interface supports up to 256 megabytes per second. The transfer rate of the interface is shared between all disks attached to the interface.

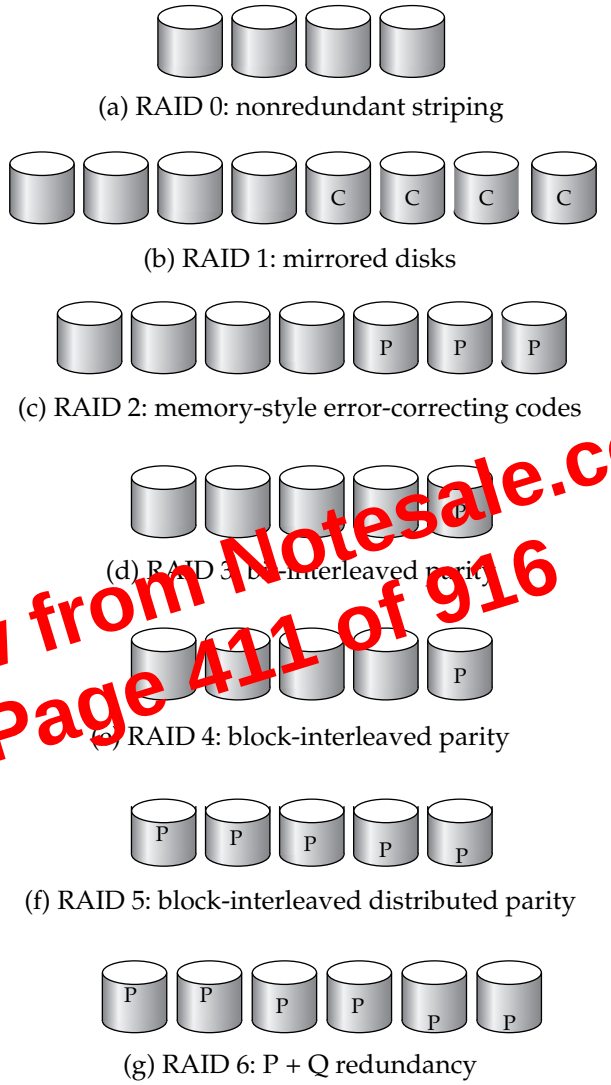


### 11.2.3 Optimization of Disk-Block Access

Requests for disk I/O are generated both by the file system and by the virtual memory manager found in most operating systems. Each request specifies the address on the disk to be referenced; that address is in the form of a *block number*. A **block** is a contiguous sequence of sectors from a single track of one platter. Block sizes range from 512 bytes to several kilobytes. Data are transferred between disk and main memory in units of blocks. The lower levels of the file-system manager convert block addresses into the hardware-level cylinder, surface, and sector number.

Since access to data on disk is several orders of magnitude slower than access to data in main memory, equipment designers have focused on techniques for improving the speed of access to blocks on disk. One such technique, buffering of blocks in memory to satisfy future requests, is discussed in Section 11.5. Here, we discuss several other techniques.

- **Scheduling**. If several blocks from a cylinder need to be transferred from disk to main memory, we may be able to save access time by requesting the blocks in the order in which they will pass under the heads. If the desired blocks are on different cylinders, it is advantageous to request the blocks in an order that minimizes disk-arm movement. **Disk-arm–scheduling** algorithms attempt to order accesses to tracks in a fashion that increases the number of accesses that can be processed. A commonly used algorithm is the **elevator algorithm**, which works in the same way many elevators do. Suppose that, initially, the arm is moving from the innermost track toward the outside of the disk. Under the elevator algorithms control, for each track for which there




(a) RAID 0: nonredundant striping

(b) RAID 1: mirrored disks

(c) RAID 2: memory-style error-correcting codes

(d) RAID 3: bit-interleaved parity

(e) RAID 4: block-interleaved parity

(f) RAID 5: block-interleaved distributed parity

(g) RAID 6: P + Q redundancy

**Figure 11.4** RAID levels.



- **RAID level 3**, bit-interleaved parity organization, improves on level 2 by exploiting the fact that disk controllers, unlike memory systems, can detect whether a sector has been read correctly, so a single parity bit can be used for error correction, as well as for detection. The idea is as follows. If one of the sectors gets damaged, the system knows exactly which sector it is, and, for each bit in the sector, the system can figure out whether it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.

RAID level 3 is as good as level 2, but is less expensive in the number of extra disks (it has only a one-disk overhead), so level 2 is not used in practice. Figure 11.4d shows the level 3 scheme.

RAID level 3 has two benefits over level 1. It needs only one parity disk for several regular disks, whereas Level 1 needs one mirror disk for every disk, and thus reduces the storage overhead. Since reads and writes of a byte are spread out over multiple disks, with $N$-way striping of data, the transfer rate for reading or writing a single block is $N$ times faster than a RAID level 1 organization using $N$-way striping. On the other hand, RAID level 3 supports a lower number of I/O operations per second, since every disk has to participate in every I/O request.

- **RAID level 4**, block-interleaved parity organization, uses block-level striping, like RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from $N$ other disks. This scheme is shown pictorially in Figure 11.4e. If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.

  A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate. The transfer rates for large reads is high, since all the disks can be read in parallel; large writes also have high transfer rates, since the data and parity can be written in parallel.

  Small independent writes, on the other hand, cannot be performed in parallel. A write of a block has to access the disk on which the block is stored, as well as the parity disk, since the parity block has to be updated. Moreover, both the old value of the parity block and the old value of the block being written have to be read for the new parity to be computed. Thus, a single write requires four disk accesses: two to read the two old blocks, and two to write the two blocks.

- **RAID level 5**, block-interleaved distributed parity, improves on level 4 by partitioning data and parity among all $N + 1$ disks, instead of storing data in $N$ disks and parity in one disk. In level 5, all disks can participate in satisfying read requests, unlike RAID level 4, where the parity disk cannot participate, so level 5 increases the total number of requests that can be met in a given amount of time. For each set of $N$ logical blocks, one of the disks stores the parity, and the other $N$ disks store the blocks.

  Figure 11.4f shows the setup. The $P$'s are distributed across all the disks. For example, with an array of 5 disks, the parity block, labelled $Pk$, for logical blocks $4k, 4k+1, 4k+2, 4k+3$ is stored in disk $(k \bmod 5) + 1$; the corresponding blocks of the other four disks store the 4 data blocks $4k$ to $4k + 3$. The following table indicates how the first 20 blocks, numbered 0 to 19, and their parity blocks are laid out. The pattern shown gets repeated on further blocks.

```
for each tuple b of borrower do
    for each tuple c of customer do
        if b[customer-name] = c[customer-name]
        then begin
                let x be a tuple defined as follows:
                x[customer-name] := b[customer-name]
                x[loan-number] := b[loan-number]
                x[customer-street] := c[customer-street]
                x[customer-city] := c[customer-city]
                include tuple x as part of result of borrower ⋈ customer
             end
    end
end
```

**Figure 11.5** Procedure for computing join.



which blocks will be referenced. Therefore, operating systems use the past pattern of block references as a predictor of future references. The assumption generally made is that blocks that have been referenced recently are likely to be referenced again. Therefore, if a block must be replaced, the least recently referenced block is replaced. This approach is called the **least recently used** (**LRU**) block-replacement scheme.

LRU is an acceptable replacement scheme in operating systems. However, a database system is able to predict the pattern of future references more accurately than an operating system. A user request to the database system involves several steps. The database system is often able to determine in advance which blocks will be needed by looking at each of the steps required to perform the user-requested operation. Thus, unlike operating systems, which must rely on the past to predict the future, database systems may have information regarding at least the short-term future.

To illustrate how information about future block access allows us to improve the LRU strategy, consider the processing of the relational-algebra expression

$$borrower \bowtie customer$$

Assume that the strategy chosen to process this request is given by the pseudocode program shown in Figure 11.5. (We shall study other strategies in Chapter 13.)

Assume that the two relations of this example are stored in separate files. In this example, we can see that, once a tuple of *borrower* has been processed, that tuple is not needed again. Therefore, once processing of an entire block of *borrower* tuples is completed, that block is no longer needed in main memory, even though it has been used recently. The buffer manager should be instructed to free the space occupied by a *borrower* block as soon as the final tuple has been processed. This buffer-management strategy is called the **toss-immediate** strategy.

Now consider blocks containing *customer* tuples. We need to examine every block of *customer* tuples once for each tuple of the *borrower* relation. When processing of a *customer* block is completed, we know that that block will not be accessed again until all other *customer* blocks have been processed. Thus, the most recently used *customer* block will be the final block to be re-referenced, and the least recently used

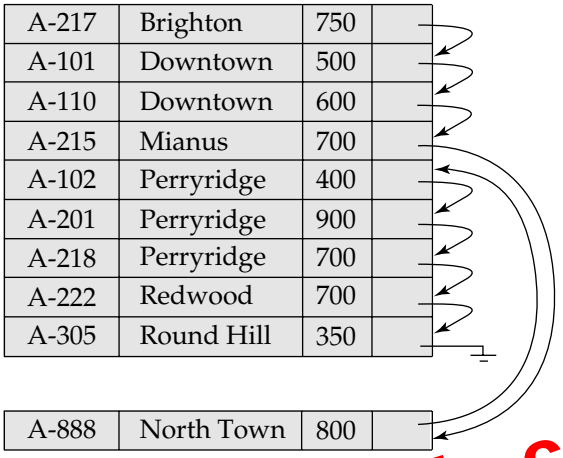



**Figure 11.16** Sequential file after an insertion.



insertion or deletion. We can manage deletion by using pointer chains, as we saw previously. For insertion, we apply the following rules:

1. Locate the record in the file that comes before the record to be inserted in search-key order.
2. If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an *overflow block*. In either case, adjust the pointers so as to chain together the records in search-key order.

Figure 11.16 shows the file of Figure 11.15 after the insertion of the record (North Town, A-888, 800). The structure in Figure 11.16 allows fast insertion of new records, but forces sequential file-processing applications to process records in an order that does not match the physical order of the records.

If relatively few records need to be stored in overflow blocks, this approach works well. Eventually, however, the correspondence between search-key order and physical order may be totally lost, in which case sequential processing will become much less efficient. At this point, the file should be **reorganized** so that it is once again physically in sequential order. Such reorganizations are costly, and must be done during times when the system load is low. The frequency with which reorganizations are needed depends on the frequency of insertion of new records. In the extreme case in which insertions rarely occur, it is possible always to keep the file in physically sorted order. In such a case, the pointer field in Figure 11.15 is not needed.

## 11.7.2 Clustering File Organization

Many relational-database systems store each relation in a separate file, so that they can take full advantage of the file system that the operating system provides. Usually, tuples of a relation can be represented as fixed-length records. Thus, relations

can be mapped to a simple file structure. This simple implementation of a relational database system is well suited to low-cost database implementations as in, for example, embedded systems or portable devices. In such systems, the size of the database is small, so little is gained from a sophisticated file structure. Furthermore, in such environments, it is essential that the overall size of the object code for the database system be small. A simple file structure reduces the amount of code needed to implement the system.

This simple approach to relational-database implementation becomes less satisfactory as the size of the database increases. We have seen that there are performance advantages to be gained from careful assignment of records to blocks, and from careful organization of the blocks themselves. Clearly, a more complicated file structure may be beneficial, even if we retain the strategy of storing each relation in a separate file.

However, many large-scale database systems do not rely directly on the underlying operating system for file management. Instead, one large operating-system file is allocated to the database system. The database system stores all relations in this one file, and manages the file itself. To see the disadvantage of storing many relations in one file, consider the following SQL query for the bank database:

**select** *account-number, customer-name, customer-street, customer-city*
**from** *depositor, customer*
**where** *depositor.customer-name = customer.customer-name*



This query computes a join of the *depositor* and *customer* relations. Thus, for each tuple of *depositor*, the system must locate the *customer* tuples with the same value for *customer-name*. Ideally, these records will be located with the help of *indices*, which we shall discuss in Chapter 12. Regardless of how these records are located, however, they need to be transferred from disk into main memory. In the worst case, each record will reside on a different block, forcing us to do one block read for each record required by the query.

As a concrete example, consider the *depositor* and *customer* relations of Figures 11.17 and 11.18, respectively. In Figure 11.19, we show a file structure designed for efficient execution of queries involving $depositor \bowtie customer$. The *depositor* tuples for each *customer-name* are stored near the *customer* tuple for the corresponding *customer-name*. This structure mixes together tuples of two relations, but allows for efficient processing of the join. When a tuple of the *customer* relation is read, the entire block containing that tuple is copied from disk into main memory. Since the corresponding

| *customer-name* | *account-number* |
|---|---|
| Hayes | A-102 |
| Hayes | A-220 |
| Hayes | A-503 |
| Turner | A-305 |

**Figure 11.17** The *depositor* relation.

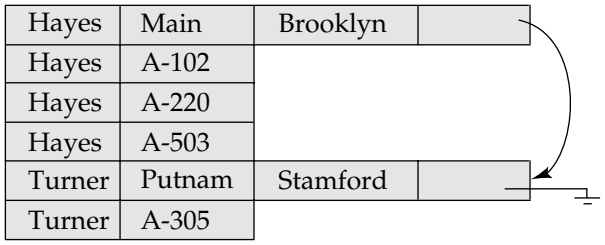



**Figure 11.20** Clustering file structure with pointer chains.

schema of the relations. This information is called the **data dictionary**, or **system catalog**. Among the types of information that the system must store are the following:

- Names of the relations
- Names of the attributes of each relation
- Domains and lengths of attributes
- Names of views defined on the database, and definitions of those views
- Integrity constraints (for example, key constraints)

In addition, many systems keep the following data on users of the system:

- Names of authorized users
- Accounting information about users
- Passwords or other information used to authenticate users

Further, the database may store statistical and descriptive data about the relations, such as:

- Number of tuples in each relation
- Method of storage for each relation (for example, clustered or nonclustered)

The data dictionary may also note the storage organization (sequential, hash or heap) of relations, and the location where each relation is stored:

- If relations are stored in operating system files, the dictionary would note the names of the file (or files) containing each relation.
- If the database stores all relations in a single file, the dictionary may note the blocks containing records of each relation in a data structure such as a linked list.

In Chapter 12, in which we study indices, we shall see a need to store information about each index on each of the relations:

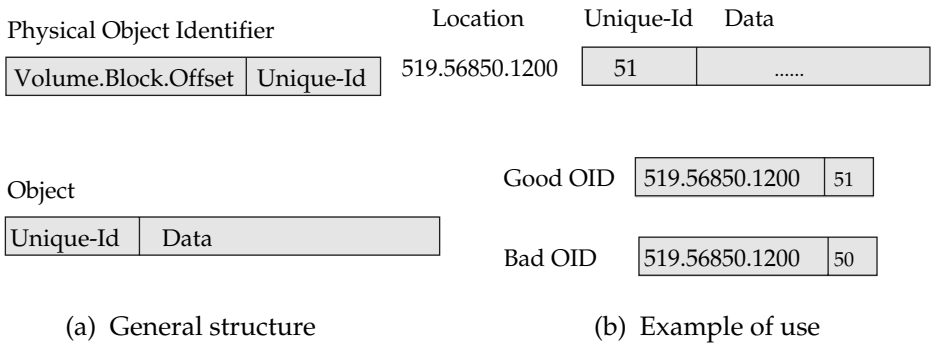



**Figure 11.21** Unique identifiers in an OID.



Suppose that an object has to be moved to a new block (perhaps because the size of the object has increased, and the old block has no extra space). Then, the physical OID will point to the old block, which no longer contains the object. Rather than change the OID of the object (which involves changing every object that points to this one), we leave behind a **forwarding address** at the old location. When the database tries to locate the object, it finds the forwarding address instead of the object; it then uses the forwarding address to locate the object.

### 11.9.3 Management of Persistent Pointers

We implement persistent pointers in a persistent programming language by using OIDs. In some implementations, persistent pointers are physical OIDs; in others, they are logical OIDs. An important difference between persistent pointers and in-memory pointers is the size of the pointer. In-memory pointers need to be only big enough to address all virtual memory. On most current computers, in-memory pointers are usually 4 bytes long, which is sufficient to address 4 gigabytes of memory. The most recent computer architectures have pointers that are 8 bytes long,

Persistent pointers need to address all the data in a database. Since database systems are often bigger than 4 gigabytes, persistent pointers are usually at least 8 bytes long. Many object-oriented databases also provide unique identifiers in persistent pointers, to catch dangling references. This feature further increases the size of persistent pointers. Thus, persistent pointers may be substantially longer than in-memory pointers.

The action of looking up an object, given its identifier, is called **dereferencing**. Given an in-memory pointer (as in C++), looking up the object is merely a memory reference. Given a persistent pointer, dereferencing an object has an extra step—finding the actual location of the object in memory by looking up the persistent pointer in a table. If the object is not already in memory, it has to be loaded from disk. We can implement the table lookup fairly efficiently by using a hash table data structure, but the lookup is still slow compared to a pointer dereference, even if the object is already in memory.

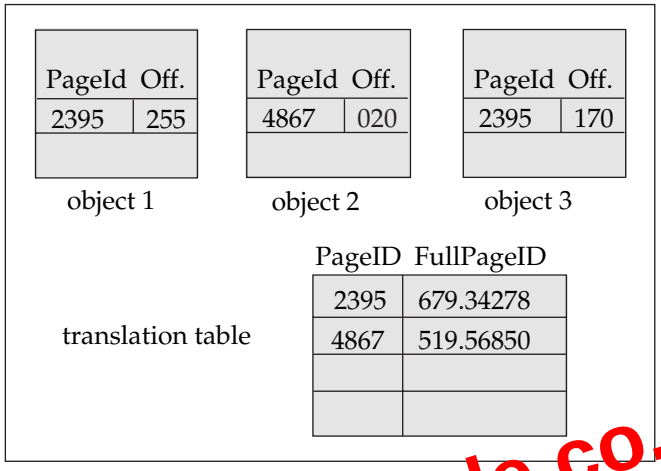



**Figure 11.22** Page image before swizzling.



#### 11.9.4.2 Swizzling Pointers on a Page

Initially no page of the database has been allocated a page in virtual memory. Virtual-memory pages may be allocated to database pages even before they are actually loaded, as we will see shortly. Database pages get loaded into virtual-memory when the database system needs to access data on the page. Before a database page is loaded, the system allocates a virtual-memory page to the database page if one has not already been allocated. The system then loads the database page into the virtual-memory page it has allocated to it.

When the system loads a database page $P$ into virtual memory, it does pointer swizzling on the page: It locates all persistent pointers contained in objects in page $P$, using the extra information stored in the page. It takes the following actions for each persistent pointer in the page. (Let the value of the persistent pointer be $\langle p_i, o_i \rangle$, where $p_i$ is the short page identifier and $o_i$ is the offset within the page. Let $P_i$ be the full page identifier of $p_i$, found in the translation table in page $P$.)

1. If page $P_i$ does not already have a virtual-memory page allocated to it, the system now allocates a free page in virtual memory to it. The page $P_i$ will reside at this virtual-memory location if and when it is brought in. At this point, the page in virtual address space does not have any storage allocated for it, either in memory or on disk; it is merely a range of addresses reserved for the database page. The system allocates actual space when it actually loads the database page $P_i$ into virtual memory.

2. Let the virtual-memory page allocated (either earlier or in the preceding step) for $P_i$ be $v_i$. The system updates the persistent pointer being considered, whose value is $\langle p_i, o_i \rangle$, by replacing $p_i$ with $v_i$.

3. It carries out pointer swizzling out on page $P_i$, as described earlier in "Swizzling Pointer on a Page".

4. After swizzling all persistent pointers in $P$, the system allows the pointer dereference that resulted in the segmentation violation to continue. The pointer dereference will find the object for which it was looking loaded in memory.

If any swizzled pointer that points to an object in page $v_i$ is dereferenced later, the dereference proceeds just like any other virtual-memory access, with no extra overheads. In contrast, if swizzling is not used, there is considerable overhead in locating the buffer page containing the object and then accessing it. This overhead has to be incurred on *every* access to objects in the page, whereas when swizzling is performed, the overhead is incurred only on the *first* access to an object in the page. Later accesses operate at regular virtual-memory access speeds. Hardware swizzling thus gives excellent performance benefits to applications that repeatedly dereference pointers.



#### 11.9.4.4 Optimizations

Software swizzling performs deswizzling operation when a page in memory has to be written back to the database, to convert in-memory pointers back to persistent pointers. Hardware swizzling can avoid this step—when the system does pointer swizzling for the page, it simply updates the translation table for the page, so that the page-identifier part of the swizzled in-memory pointers can be used to look up the table. For example, as shown in Figure 11.23, database page 679.34278 (with short identifier 2395 in the page shown) is mapped to virtual-memory page 5001. At this point, not only is the pointer in object 1 updated from 2395255 to 5001255, but also the short identifier in the table is updated to 5001. Thus, the short identifier 5001 in object 1 and in the table match each other again. Therefore, the page can be written back to disk without any deswizzling.

Several optimizations can be carried out on the basic scheme described here. When the system swizzles page $P$, for each page $P'$ referred to by any persistent pointer in $P$, it attempts to allocate $P'$ to the virtual address location indicated by the short page identifier of $P'$ on page $P$. If the system can allocate the page in this attempt, pointers to it do not need to be updated. In our swizzling example, page 519.56850 with short page identifier 4867 was mapped to virtual-memory page 4867, which is the same as its short page identifier. We can see that the pointer in object 2 to this page did not need to be changed during swizzling. If every page can be allocated to its appropriate location in virtual address space, none of the pointers need to be translated, and the cost of swizzling is reduced significantly.

Hardware swizzling works even if the database is bigger than virtual memory, but only as long as all the pages that a particular process accesses fit into the virtual memory of the process. If they do not, a page that has been brought into virtual memory will have to be replaced, and that replacement is hard to do, since there may be in-memory pointers to objects in that page.

**11.4** A power failure that occurs while a disk block is being written could result in the block being only partially written. Assume that partially written blocks can be detected. An atomic block write is one where either the disk block is fully written or nothing is written (i.e., there are no partial writes). Suggest schemes for getting the effect of atomic block writes with the following RAID schemes. Your schemes should involve work on recovery from failure.

**a.** RAID level 1 (mirroring)
**b.** RAID level 5 (block interleaved, distributed parity)

**11.5** RAID systems typically allow you to replace failed disks without stopping access to the system. Thus, the data in the failed disk must be rebuilt and written to the replacement disk while the system is in operation. With which of the RAID levels is the amount of interference between the rebuild and ongoing disk accesses least? Explain your answer.

**11.6** Give an example of a relational-algebra expression and a query-processing strategy in each of the following situations:

**a.** MRU is preferable to LRU.
**b.** LRU is preferable to MRU.

**11.7** Consider the deletion of record 5 from the file of Figure 11.8. Compare the relative merits of the following techniques for implementing the deletion:

**a.** Move record 6 to the space occupied by record 5, and move record 7 to the space occupied by record 6.
**b.** Move record 7 to the space occupied by record 5.
**c.** Mark record 5 as deleted, and move no records.

**11.8** Show the structure of the file of Figure 11.9 after each of the following steps:

**a.** Insert (Brighton, A-323, 1600).
**b.** Delete record 2.
**c.** Insert (Brighton, A-626, 2000).

**11.9** Give an example of a database application in which the reserved-space method of representing variable-length records is preferable to the pointer method. Explain your answer.

**11.10** Give an example of a database application in which the pointer method of representing variable-length records is preferable to the reserved-space method. Explain your answer.

**11.11** Show the structure of the file of Figure 11.12 after each of the following steps:

**a.** Insert (Mianus, A-101, 2800).
**b.** Insert (Brighton, A-323, 1600).
**c.** Delete (Perryridge, A-102, 400).

**11.22** If physical OIDs are used, an object can be relocated by keeping a forwarding pointer to its new location. In case an object gets forwarded multiple times, what would be the effect on retrieval speed? Suggest a technique to avoid multiple accesses in such a case.

**11.23** Define the term *dangling pointer*. Describe how the unique-id scheme helps in detecting dangling pointers in an object-oriented database.

**11.24** Consider the example on page 435, which shows that there is no need for deswizzling if hardware swizzling is used. Explain why, in that example, it is safe to change the short identifier of page 679.34278 from 2395 to 5001. Can some other page already have short identifier 5001? If it could, how can you handle that situation?



## Bibliographical Notes

Patterson and Hennessy [1995] discusses the hardware aspects of translation look-aside buffers, caches, and memory-management units. Tosch and Wethington [1999] presents an excellent overview of computer hardware, including extensive coverage of all types of storage technology, such as floppy disks, magnetic disks, optical disks, tapes, and storage interfaces. Ruemmler and Wilkes [1994] presents a survey of magnetic-disk technology. Flash memory is discussed by Dippert and Levy [1993].

The specifications of current-generation disk drives can be obtained from the Web sites of their manufacturers, such as IBM, Seagate, and Maxtor.

Alternative disk organizations that provide a high degree of fault tolerance include those described by Gray et al. [1990] and Bitton and Gray [1988]. Disk striping is described by Salem and Garcia-Molina [1986]. Discussions of redundant arrays of inexpensive disks (RAID) are presented by Patterson et al. [1988] and Chen and Patterson [1990]. Chen et al. [1994] presents an excellent survey of RAID principles and implementation. Reed–Solomon codes are covered in Pless [1989]. The log-based file system, which makes disk access sequential, is described in Rosenblum and Ousterhout [1991].

In systems that support mobile computing, data may be broadcast repeatedly. The broadcast medium can be viewed as a level of the storage hierarchy—as a broadcast disk with high latency. These issues are discussed in Acharya et al. [1995]. Caching and buffer management for mobile computing is discussed in Barbará and Imielinski [1994]. Further discussion of storage issues in mobile computing appears in Douglis et al. [1994].

Basic data structures are discussed in Cormen et al. [1990]. There are several papers describing the storage structure of specific database systems. Astrahan et al. [1976] discusses System R. Chamberlin et al. [1981] reviews System R in retrospect. The *Oracle 8 Concepts Manual* (Oracle [1997]) describes the storage organization of the Oracle 8 database system. The structure of the Wisconsin Storage System (WiSS) is described in Chou et al. [1985]. A software tool for the physical design of relational databases is described by Finkelstein et al. [1988].

index is a multilevel index: The words at the top of each page of the book index form a sparse index on the contents of the dictionary pages.

Multilevel indices are closely related to tree structures, such as the binary trees used for in-memory indexing. We shall examine the relationship later, in Section 12.3.

#### 12.2.1.3 Index Update

Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file. We first describe algorithms for updating single-level indices.

- **Insertion**. First, the system performs a lookup using the search-key value that appears in the record to be inserted. Again, the actions the system takes next depend on whether the index is dense or sparse:
  - Dense indices:
    1. If the search-key value does not appear in the index, the system inserts an index record with the search-key value in the index at the appropriate position.
    2. Otherwise the following actions are taken:
       a. If the index record stores pointers to all records with the same search-key value, the system adds a pointer to the new record to the index record.
       b. Otherwise, the index record stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.
  - Sparse indices: We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into the index. On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.
- **Deletion**. To delete a record, the system first looks up the record to be deleted. The actions the system takes next depend on whether the index is dense or sparse:
  - Dense indices:
    1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index record from the index.
    2. Otherwise the following actions are taken:
       a. If the index record stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index record.

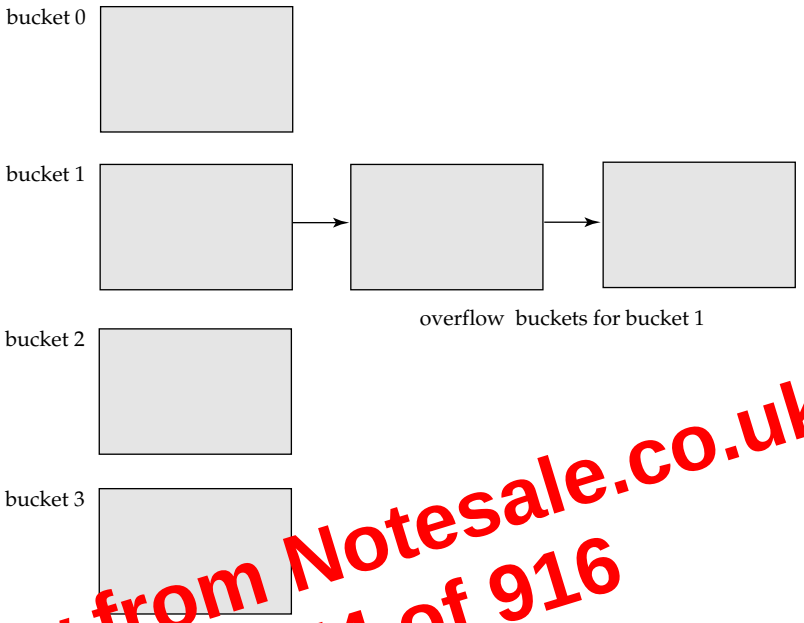



**Figure 12.22** Overflow chaining in a hash structure.



ets of a given bucket are chained together in a linked list, as in Figure 12.22. Overflow handling using such a linked list is called **overflow chaining**.

We must change the lookup algorithm slightly to handle overflow chaining. As before, the system uses the hash function on the search key to identify a bucket $b$. The system must examine all the records in bucket $b$ to see whether they match the search key, as before. In addition, if bucket $b$ has overflow buckets, the system must examine the records in all the overflow buckets also.

The form of hash structure that we have just described is sometimes referred to as **closed hashing**. Under an alternative approach, called **open hashing**, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets $B$. One policy is to use the next bucket (in cyclic order) that has space; this policy is called *linear probing*. Other policies, such as computing further hash functions, are also used. Open hashing has been used to construct symbol tables for compilers and assemblers, but closed hashing is preferable for database systems. The reason is that deletion under open hashing is troublesome. Usually, compilers and assemblers perform only lookup and insertion operations on their symbol tables. However, in a database system, it is important to be able to handle deletion as well as insertion. Thus, open hashing is of only minor importance in database implementation.

An important drawback to the form of hashing that we have described is that we must choose the hash function when we implement the system, and it cannot be changed easily thereafter if the file being indexed grows or shrinks. Since the function $h$ maps search-key values to a fixed set $B$ of bucket addresses, we waste space if $B$ is

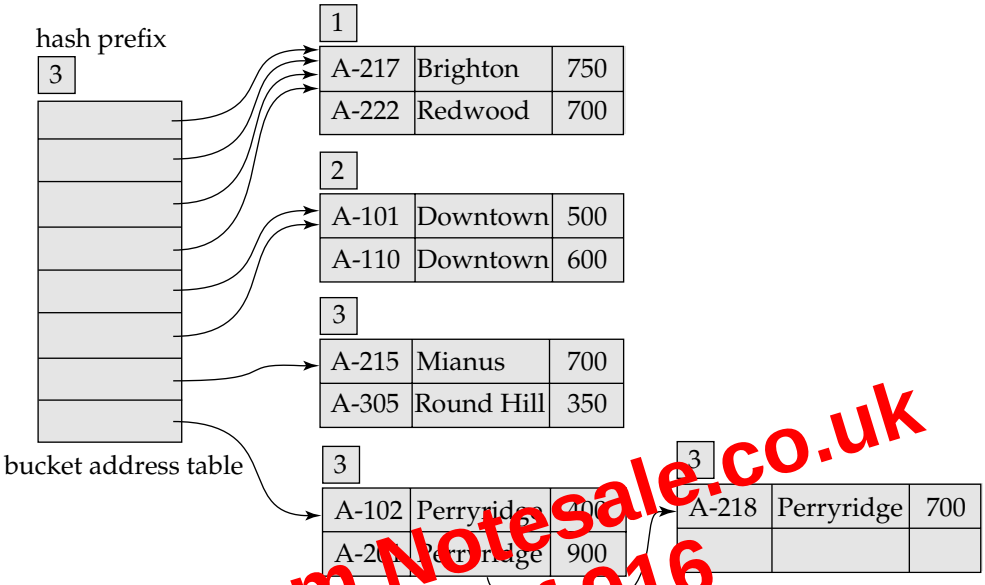





**Figure 12.31** Extendable hash structure for the *account* file.

fix length. Thus table is thus small. The main space saving of extendable hashing over other forms of hashing is that no buckets need to be reserved for future growth; rather, buckets can be allocated dynamically.

A disadvantage of extendable hashing is that lookup involves an additional level of indirection, since the system must access the bucket address table before accessing the bucket itself. This extra reference has only a minor effect on performance. Although the hash structures that we discussed in Section 12.5 do not have this extra level of indirection, they lose their minor performance advantage as they become full.

Thus, extendable hashing appears to be a highly attractive technique, provided that we are willing to accept the added complexity involved in its implementation. The bibliographical notes reference more detailed descriptions of extendable hashing implementation. The bibliographical notes also provide references to another form of dynamic hashing called **linear hashing**, which avoids the extra level of indirection associated with extendable hashing, at the possible cost of more overflow buckets.

## 12.7 Comparison of Ordered Indexing and Hashing

We have seen several ordered-indexing schemes and several hashing schemes. We can organize files of records as ordered files, by using index-sequential organization or $B^+$-tree organizations. Alternatively, we can organize the files by using hashing. Finally, we can organize them as heap files, where the records are not ordered in any particular way.

**create index** *b-index* **on** *branch* (*branch-name*)

If we wish to declare that the search key is a candidate key, we add the attribute **unique** to the index definition. Thus, the command

**create unique index** *b-index* **on** *branch* (*branch-name*)

declares *branch-name* to be a candidate key for *branch*. If, at the time we enter the **create unique index** command, *branch-name* is not a candidate key, the system will display an error message, and the attempt to create the index will fail. If the index-creation attempt succeeds, any subsequent attempt to insert a tuple that violates the key declaration will fail. Note that the **unique** feature is redundant if the database system supports the **unique** declaration of the SQL standard.

Many database systems also provide a way to specify the type of index to be used (such as $B^+$-tree or hashing). Some database systems also permit one of the indices on a relation to be declared to be clustered; the system then stores the relation sorted by the search-key of the clustered index.

The index name we specified for an index is required to drop an index. The **drop index** command takes the form:

**drop index** <index-name>



## 12.9 Multiple-Key Access

Until now, we have assumed implicitly that only one index (or hash table) is used to process a query on a relation. However, for certain types of queries, it is advantageous to use multiple indices if they exist.

### 12.9.1 Using Multiple Single-Key Indices

Assume that the *account* file has two indices: one for *branch-name* and one for *balance*. Consider the following query: "Find all account numbers at the Perryridge branch with balances equal to $1000." We write

**select** *loan-number*
**from** *account*
**where** *branch-name* = "Perryridge" **and** *balance* = 1000

There are three strategies possible for processing this query:

1. Use the index on *branch-name* to find all records pertaining to the Perryridge branch. Examine each such record to see whether *balance* = 1000.
2. Use the index on *balance* to find all records pertaining to accounts with balances of $1000. Examine each such record to see whether *branch-name* = "Perryridge."
3. Use the index on *branch-name* to find *pointers* to all records pertaining to the Perryridge branch. Also, use the index on *balance* to find pointers to all records

- The primary disadvantage of the index-sequential file organization is that performance degrades as the file grows. To overcome this deficiency, we can use a $B^+$*-tree index*.
- A $B^+$-tree index takes the form of a *balanced* tree, in which every path from the root of the tree to a leaf of the tree is of the same length. The height of a $B^+$-tree is proportional to the logarithm to the base $N$ of the number of records in the relation, where each nonleaf node stores $N$ pointers; the value of $N$ is often around 50 or 100. $B^+$-trees are much shorter than other balanced binary-tree structures such as AVL trees, and therefore require fewer disk accesses to locate records.
- Lookup on $B^+$-trees is straightforward and efficient. Insertion and deletion, however, are somewhat more complicated, but still efficient. The number of operations required for lookup, insertion, and deletion on $B^+$-trees is proportional to the logarithm to the base $N$ of the number of records in the relation, where each nonleaf node stores $N$ pointers.
- We can use $B^+$-trees for indexing a file containing records, as well as to organize records within a file.
- B-tree indices are similar to $B^+$-tree indices. The primary advantage of a B-tree is that the B-tree eliminates the redundant storage of search-key values. The major disadvantages are overall complexity and reduced fanout for a given node size. System designers almost universally prefer $B^+$-tree indices over B-tree indices in practice.
- Sequential file organizations require an index structure to locate data. File organizations based on hashing, by contrast, allow us to find the address of a data item directly by computing a function on the search-key value of the desired record. Since we do not know at design time precisely which search-key values will be stored in the file, a good hash function to choose is one that assigns search-key values to buckets such that the distribution is both uniform and random.
- *Static hashing* uses hash functions in which the set of bucket addresses is fixed. Such hash functions cannot easily accommodate databases that grow significantly larger over time. There are several *dynamic hashing techniques* that allow the hash function to be modified. One example is *extendable hashing*, which copes with changes in database size by splitting and coalescing buckets as the database grows and shrinks.
- We can also use hashing to create secondary indices; such indices are called *hash indices*. For notational convenience, we assume hash file organizations have an implicit hash index on the search key used for hashing.
- Ordered indices such as $B^+$-trees and hash indices can be used for selections based on equality conditions involving single attributes. When multiple

**12.14** Give pseudocode for deletion of entries from an extendable hash structure, including details of when and how to coalesce buckets. Do not bother about reducing the size of the bucket address table.

**12.15** Suggest an efficient way to test if the bucket address table in extendable hashing can be reduced in size, by storing an extra count with the bucket address table. Give details of how the count should be maintained when buckets are split, coalesced or deleted.

(Note: Reducing the size of the bucket address table is an expensive operation, and subsequent inserts may cause the table to grow again. Therefore, it is best not to reduce the size as soon as it is possible to do so, but instead do it only if the number of index entries becomes small compared to the bucket address table size.)

**12.16** Why is a hash structure not the best choice for a search key on which range queries are likely?

**12.17** Consider a grid file in which we wish to avoid overflow buckets for performance reasons. In cases where an overflow bucket would be needed, we instead reorganize the grid file. Present an algorithm for such a reorganization.

**12.18** Consider the *account* relation shown in Figure 12.25.

**a.** Construct a bitmap index on the attributes *branch-name* and *balance*, dividing *balance* values into 4 ranges: below 250, 250 to below 500, 500 to below 750, and 750 and above.

**b.** Consider a query that requests all accounts in Downtown with a balance of 500 or more. Outline the steps in answering the query, and show the final and intermediate bitmaps constructed to answer the query.

**12.19** Show how to compute existence bitmaps from other bitmaps. Make sure that your technique works even in the presence of null values, by using a bitmap for the value *null*.

**12.20** How does data encryption affect index schemes? In particular, how might it affect schemes that attempt to store data in sorted order?



## Bibliographical Notes

Discussions of the basic data structures in indexing and hashing can be found in Cormen et al. [1990]. B-tree indices were first introduced in Bayer [1972] and Bayer and McCreight [1972]. $B^+$-trees are discussed in Comer [1979], Bayer and Unterauer [1977] and Knuth [1973]. The bibliographic notes in Chapter 16 provides references to research on allowing concurrent accesses and updates on $B^+$-trees. Gray and Reuter [1993] provide a good description of issues in the implementation of $B^+$-trees.

Several alternative tree and treelike search structures have been proposed. **Tries** are trees whose structure is based on the "digits" of keys (for example, a dictionary thumb index, which has one entry for each letter). Such trees may not be balanced in the sense that $B^+$-trees are. Tries are discussed by Ramesh et al. [1989], Orenstein

In relational systems, a file scan allows an entire relation to be read in those cases where the relation is stored in a single, dedicated file.

### 13.3.1 Basic Algorithms

Consider a selection operation on a relation whose tuples are stored together in one file. Two scan algorithms to implement the selection operation are:

- **A1 (linear search).** In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition. For a selection on a key attribute, the system can terminate the scan if the required record is found, without looking at the other records of the relation.

  The cost of linear search, in terms of number of I/O operations, is $b_r$, where $b_r$ denotes the number of blocks in the file. Selections on key attributes have an average cost of $b_r/2$, but still have a worst-case cost of $b_r$.

  Although it may be slower than other algorithms for implementing selection, the linear search algorithm can be applied to any file, regardless of the ordering of the file, or the availability of indices, or the nature of the selection operation. The other algorithms that we shall study are not applicable in all cases, but when applicable they are generally faster than linear search.

- **A2 (binary search).** If the file is ordered on an attribute, and the selection condition is an equality comparison on the attribute, we can use a binary search to locate records that satisfy the selection. The system performs the binary search on the blocks of the file.

  The number of blocks that need to be examined to find a block containing the required records is $\lceil \log_2(b_r) \rceil$, where $b_r$ denotes the number of blocks in the file. If the selection is on a nonkey attribute, more than one block may contain required records, and the cost of reading the extra blocks has to be added to the cost estimate. We can estimate this number by estimating the size of the selection result (which we cover in Section 14.2), and dividing it by the average number of records that are stored per block of the relation.



### 13.3.2 Selections Using Indices

Index structures are referred to as **access paths**, since they provide a path through which data can be located and accessed. In Chapter 12, we pointed out that it is efficient to read the records of a file in an order corresponding closely to physical order. Recall that a *primary index* is an index that allows the records of a file to be read in an order that corresponds to the physical order in the file. An index that is not a primary index is called a *secondary index*.

Search algorithms that use an index are referred to as **index scans**. Ordered indices, such as $B^+$-trees, also permit access to tuples in a sorted order, which is useful for implementing range queries. Although indices can provide fast, direct, and ordered access, they impose the overhead of access to those blocks containing the index. We use the selection predicate to guide us in the choice of the index to use in processing the query. Search algorithms that use an index are:

We compute how many block transfers are required for the external sort merge in this way: Let $b_r$ denote the number of blocks containing records of relation $r$. The first stage reads every block of the relation and writes them out again, giving a total of $2b_r$ disk accesses. The initial number of runs is $\lceil b_r/M \rceil$. Since the number of runs decreases by a factor of $M-1$ in each merge pass, the total number of merge passes required is $\lceil \log_{M-1}(b_r/M) \rceil$. Each of these passes reads every block of the relation once and writes it out once, with two exceptions. First, the final pass can produce the sorted output without writing its result to disk. Second, there may be runs that are not read in or written out during a pass—for example, if there are $M$ runs to be merged in a pass, $M-1$ are read in and merged, and one run is not accessed during the pass. Ignoring the (relatively small) savings due to the latter effect, the total number of disk accesses for external sorting of the relation is

$$b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$$

Applying this equation to the example in Figure 13.3 we get a total of $12*(4+1) = 60$ block transfers, as you can verify from the figure. Note that this value does not include the cost of writing out the final result.



## 13.5 Join Operation

In this section, we study several algorithms for computing the join of relations, and we analyze their respective costs.

We use the term **equi-join** to refer to a join of the form $r \bowtie_{r.A=s.B} s$, where $A$ and $B$ are attributes or sets of attributes of relations $r$ and $s$ respectively.

We use as a running example the expression

$$depositor \bowtie customer$$

We assume the following information about the two relations:

- Number of records of *customer*: $n_{customer} = 10,000$.
- Number of blocks of *customer*: $b_{customer} = 400$.
- Number of records of *depositor*: $n_{depositor} = 5000$.
- Number of blocks of *depositor*: $b_{depositor} = 100$.

### 13.5.1 Nested-Loop Join

Figure 13.4 shows a simple algorithm to compute the theta join, $r \bowtie_\theta s$, of two relations $r$ and $s$. This algorithm is called the **nested-loop join** algorithm, since it basically consists of a pair of nested **for** loops. Relation $r$ is called the **outer relation** and relation $s$ the **inner relation** of the join, since the loop for $r$ encloses the loop for $s$. The algorithm uses the notation $t_r \cdot t_s$, where $t_r$ and $t_s$ are tuples; $t_r \cdot t_s$ denotes the tuple constructed by concatenating the attribute values of tuples $t_r$ and $t_s$.

Like the linear file-scan algorithm for selection, the nested-loop join algorithm requires no indices, and it can be used regardless of what the join condition is. Extending the algorithm to compute the natural join is straightforward, since the natural

```
for each tuple t_r in r do begin
    for each tuple t_s in s do begin
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add t_r · t_s to the result.
    end
end
```

**Figure 13.4** Nested-loop join.

join can be expressed as a theta join followed by elimination of repeated attributes by a projection. The only change required is an extra step of deleting repeated attributes from the tuple $t_r \cdot t_s$, before adding it to the result.

The nested-loop join algorithm is expensive, since it examines every pair of tuples in the two relations. Consider the cost of the nested-loop join algorithm. The number of pairs of tuples to be considered is $n_r * n_s$, where $n_r$ denotes the number of tuples in $r$, and $n_s$ denotes the number of tuples in $s$. For each record in $r$, we have to perform a complete scan on $s$. In the worst case, the buffer can hold only one block of each relation, and a total of $n_r * b_s + b_r$ block accesses would be required, where $b_r$ and $b_s$ denote the number of blocks containing tuples of $r$ and $s$ respectively. In the best case, there is enough space for both relations to fit simultaneously in memory, so each block would have to be read only once; hence, only $b_r + b_s$ block accesses would be required.

If one of the relations fits entirely in main memory, it is beneficial to use that relation as the inner relation, since the inner relation would then be read only once. Therefore, if $s$ is small enough to fit in main memory, our strategy requires only a total $b_r + b_s$ accesses—the same cost as that for the case where both relations fit in memory.

Now consider the natural join of *depositor* and *customer*. Assume for now that we have no indices whatsoever on either relation, and that we are not willing to create any index. We can use the nested loops to compute the join; assume that *depositor* is the outer relation and *customer* is the inner relation in the join. We will have to examine $5000 * 10000 = 50 * 10^6$ pairs of tuples. In the worst case, the number of block accesses is $5000 * 400 + 100 =$ 2,000,100. In the best-case scenario, however, we can read both relations only once, and perform the computation. This computation requires at most $100 + 400 = 500$ block accesses—a significant improvement over the worst-case scenario. If we had used *customer* as the relation for the outer loop and *depositor* for the inner loop, the worst-case cost of our final strategy would have been lower: $10000 * 100 + 400 =$ 1,000,400.



## 13.5.2 Block Nested-Loop Join

If the buffer is too small to hold either relation entirely in memory, we can still obtain a major saving in block accesses if we process the relations on a per-block basis, rather than on a per-tuple basis. Figure 13.5 shows **block nested-loop join**, which is a variant of the nested-loop join where every block of the inner relation is paired with every block of the outer relation. Within each pair of blocks, every tuple in one block

### 13.7.2.1 Implementation of Pipelining

We can implement a pipeline by constructing a single, complex operation that combines the operations that constitute the pipeline. Although this approach may be feasible for various frequently occurring situations, it is desirable in general to reuse the code for individual operations in the construction of a pipeline. Therefore, each operation in the pipeline is modeled as a separate process or thread within the system, which takes a stream of tuples from its pipelined inputs, and generates a stream of tuples for its output. For each pair of adjacent operations in the pipeline, the system creates a buffer to hold tuples being passed from one operation to the next.

In the example of Figure 13.10, all three operations can be placed in a pipeline, which passes the results of the selection to the join as they are generated. In turn, it passes the results of the join to the projection as they are generated. The memory requirements are low, since results of an operation are not stored for long. However, as a result of pipelining, the inputs to the operations are not available all at once for processing.

Pipelines can be executed in either of two ways:

1. Demand driven
2. Producer driven

In a **demand-driven pipeline**, the system makes repeated requests for tuples from the operation at the top of the pipeline. Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) to be returned, and then returns that tuple. If the inputs of the operation are not pipelined, the next tuple(s) to be returned can be computed from the input relations, while the system keeps track of what has been returned so far. If it has some pipelined inputs, the operation also makes requests for tuples from its pipelined inputs. Using the tuples received from its pipelined inputs, the operation computes tuples for its output, and passes them up to its parent.

In a **producer-driven pipeline**, operations do not wait for requests to produce tuples, but instead generate the tuples **eagerly**. Each operation at the bottom of a pipeline continually generates output tuples, and puts them in its output buffer, until the buffer is full. An operation at any other level of a pipeline generates output tuples when it gets input tuples from lower down in the pipeline, until its output buffer is full. Once the operation uses a tuple from a pipelined input, it removes the tuple from its input buffer. In either case, once the output buffer is full, the operation waits until its parent operation removes tuples from the buffer, so that the buffer has space for more tuples. At this point, the operation generates more tuples, until the buffer is full again. The operation repeats this process until all the output tuples have been generated.

It is necessary for the system to switch between operations only when an output buffer is full, or an input buffer is empty and more input tuples are needed to generate any more output tuples. In a parallel-processing system, operations in a pipeline may be run concurrently on distinct processors (see Chapter 20).

Using producer-driven pipelining can be thought of as **pushing** data up an operation tree from below, whereas using demand-driven pipelining can be thought of as

**13.9** Let $r$ and $s$ be relations with no indices, and assume that the relations are not sorted. Assuming infinite memory, what is the lowest cost way (in terms of I/O operations) to compute $r \bowtie s$? What is the amount of memory required for this algorithm?

**13.10** Suppose that a $B^+$-tree index on *branch-city* is available on relation *branch*, and that no other index is available. List different ways to handle the following selections that involve negation?

**a.** $\sigma_{\neg(branch\text{-}city < \text{“Brooklyn”})}(branch)$
**b.** $\sigma_{\neg(branch\text{-}city = \text{“Brooklyn”})}(branch)$
**c.** $\sigma_{\neg(branch\text{-}city < \text{“Brooklyn”} \vee assets < 5000)}(branch)$

**13.11** The hash join algorithm as described in Section 13.5.5 computes the natural join of two relations. Describe how to extend the hash join algorithm to compute the natural left outer join, the natural right outer join and the natural full outer join. (Hint: Keep extra information with each tuple in the hash index, to detect whether any tuple in the probe relation matches the tuple in the hash index.) Try out your algorithm on the *customer* and *depositor* relations.

**13.12** Write pseudocode for an iterator that implements indexed nested-loop join, where the outer relation is pipelined. Use the standard iterator functions in your pseudocode. Show what state information the iterator must maintain between calls.

**13.13** Design sorting based and hashing algorithms for computing the division operation.



## Bibliographical Notes

A query processor must parse statements in the query language, and must translate them into an internal form. Parsing of query languages differs little from parsing of traditional programming languages. Most compiler texts, such as Aho et al. [1986], cover the main parsing techniques, and present optimization from a programming-language point of view.

Knuth [1973] presents an excellent description of external sorting algorithms, including an optimization that can create initial runs that are (on the average) twice the size of memory. Based on performance studies conducted in the mid-1970s, database systems of that period used only nested-loop join and merge join. These studies, which were related to the development of System R, determined that either the nested-loop join or merge join nearly always provided the optimal join method (Blasgen and Eswaran [1976]); hence, these two were the only join algorithms implemented in System R. The System R study, however, did not include an analysis of hash join algorithms. Today, hash joins are considered to be highly efficient.

Hash join algorithms were initially developed for parallel database systems. Hash join techniques are described in Kitsuregawa et al. [1983], and extensions including hybrid hash join are described in Shapiro [1986]. Zeller and Gray [1990] and Davison and Graefe [1994] describe hash join techniques that can adapt to the available mem-

Recall that the natural-join operator is simply a special case of the theta-join operator; hence, natural joins are also commutative.

6. **a.** Natural-join operations are **associative**.

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

**b.** Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where $\theta_2$ involves attributes from only $E_2$ and $E_3$. Any of these conditions may be empty; hence, it follows that the Cartesian product ($\times$) operation is also associative. The commutativity and associativity of join operations are important for join reordering in query optimization.

7. The selection operation distributes over the theta-join operation under the following two conditions:

**a.** It distributes when all the attributes in selection condition $\theta_0$ involve only the attributes of one of the expressions (say, $E_1$) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

**b.** It distributes when selection condition $\theta_1$ involves only the attributes of $E_1$ and $\theta_2$ involves only the attributes of $E_2$.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

8. The projection operation distributes over the theta-join operation under the following conditions.

**a.** Let $L_1$ and $L_2$ be attributes of $E_1$ and $E_2$, respectively. Suppose that the join condition $\theta$ involves only attributes in $L_1 \cup L_2$. Then,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

**b.** Consider a join $E_1 \bowtie_\theta E_2$. Let $L_1$ and $L_2$ be sets of attributes from $E_1$ and $E_2$, respectively. Let $L_3$ be attributes of $E_1$ that are involved in join condition $\theta$, but are not in $L_1 \cup L_2$, and let $L_4$ be attributes of $E_2$ that are involved in join condition $\theta$, but are not in $L_1 \cup L_2$. Then,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

9. The set operations union and intersection are commutative.

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

Set difference is not commutative.

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

apply rule 6.a (associativity of natural join) to transform the join $branch \bowtie (account \bowtie depositor)$ into $(branch \bowtie account) \bowtie depositor$:

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}city = \text{“Brooklyn”}\ \wedge\ balance\ >1000} ((branch \bowtie\ account) \bowtie\ depositor))$$

Then, using rule 7.a, we can rewrite our query as

$$\Pi_{customer\text{-}name}\ ((\sigma_{branch\text{-}city = \text{“Brooklyn”}\wedge\ balance>1000} (branch \bowtie\ account)) \bowtie\ depositor)$$

Let us examine the selection subexpression within this expression. Using rule 1, we can break the selection into two selections, to get the following subexpression:

$$\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (\sigma_{balance\ >\ 1000}\ (branch \bowtie\ account))$$

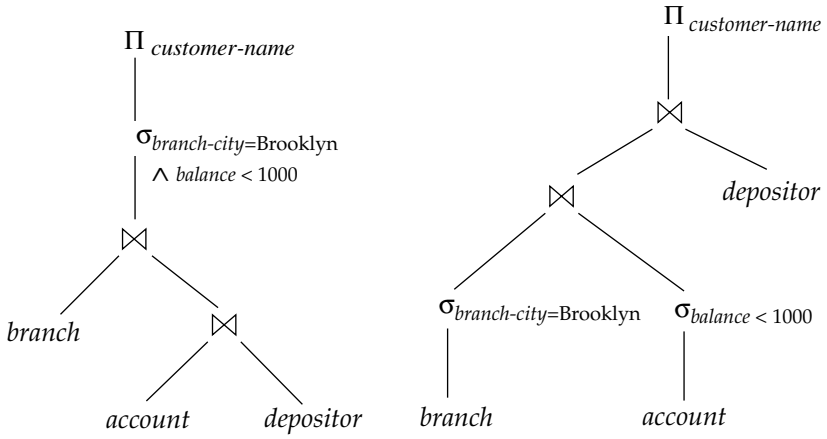
Both of the preceding expressions select tuples with *branch-city* = “Brooklyn” and *balance* $>$ 1000. However, the latter form of the expression provides a new opportunity to apply the “perform selections early” rule, resulting in the subexpression

$$\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch) \bowtie \sigma_{balance>1000}\ (account)$$

Figure 14.3 depicts the initial expression and the final expression after all these transformations. We could equally well have used rule 7.b to get the final expression directly, without using rule 1 to break the selection into two selections. In fact, rule 7.b can itself be derived from rules 1 and 7.a.

A set of equivalence rules is said to be **minimal** if no rule can be derived from any combination of the others. The preceding example illustrates that the set of equivalence rules in Section 14.3.1 is not minimal. An expression equivalent to the original expression may be generated in different ways; the number of different ways of generating an expression increases when we use a nonminimal set of equivalence rules. Query optimizers therefore use minimal sets of equivalence rules.






(a) Initial expression tree (b) Tree after multiple transformations

**Figure 14.3** Multiple transformations.

each interesting sort order of the join result for that subset. The number of subsets of $n$ relations is $2^n$. The number of interesting sort orders is generally not large. Thus, about $2^n$ join expressions need to be stored. The dynamic-programming algorithm for finding the best join order can be easily extended to handle sort orders. The cost of the extended algorithm depends on the number of interesting orders for each subset of relations; since this number has been found to be small in practice, the cost remains at $O(3^n)$.

With $n = 10$, this number is around 59000, which is much better than the 17.6 billion different join orders. More important, the storage required is much less than before, since we need to store only one join order for each interesting sort order of each of 1024 subsets of $r_1, \ldots, r_{10}$. Although both numbers still increase rapidly with $n$, commonly occurring joins usually have less than 10 relations, and can be handled easily.

We can use several techniques to reduce further the cost of searching through a large number of plans. For instance, when examining the plans for an expression, we can terminate after we examine only a part of the expression, if we determine that the cheapest plan for that part is already costlier than the cheapest evaluation plan for a full expression examined earlier. Similarly, suppose that we determine that the cheapest way of evaluating a subexpression is costlier than the cheapest evaluation plan for a full expression examined earlier. Then, no full expression involving that subexpression needs to be examined. We can further reduce the number of evaluation plans that need to be considered fully by first making a heuristic guess of a good plan, and estimating that plan's cost. Then, only a few competing plans will require a full analysis of cost. These optimizations can reduce the overhead of query optimization significantly.



### 14.4.3 Heuristic Optimization

A drawback of cost-based optimization is the cost of optimization itself. Although the cost of query processing can be reduced by clever optimizations, cost-based optimization is still expensive. Hence, many systems use **heuristics** to reduce the number of choices that must be made in a cost-based fashion. Some systems even choose to use only heuristics, and do not use cost-based optimization at all.

An example of a heuristic rule is the following rule for transforming relational-algebra queries:

- Perform selection operations as early as possible.

A heuristic optimizer would use this rule without finding out whether the cost is reduced by this transformation. In the first transformation example in Section 14.3, the selection operation was pushed into a join.

We say that the preceding rule is a heuristic because it usually, but not always, helps to reduce the cost. For an example of where it can result in an increase in cost, consider an expression $\sigma_\theta(r \bowtie s)$, where the condition $\theta$ refers to only attributes in $s$. The selection can certainly be performed before the join. However, if $r$ is extremely small compared to $s$, and if there is an index on the join attributes of $s$, but no index on the attributes used by $\theta$, then it is probably a bad idea to perform the selection

Instead, to handle the case of **avg**, we maintain the **sum** and **count** aggregate values as described earlier, and compute the average as the sum divided by the count.

- **min, max**: Consider a materialized view $v = {}_A\mathcal{G}_{min(B)}(r)$. (The case of **max** is exactly equivalent.)

  Handling insertions on $r$ is straightforward. Maintaining the aggregate values **min** and **max** on deletions may be more expensive. For example, if the tuple corresponding to the minimum value for a group is deleted from $r$, we have to look at the other tuples of $r$ that are in the same group to find the new minimum value.

#### 14.5.2.4 Other Operations

The set operation *intersection* is maintained as follows. Given materialized view $v = r \cap s$, when a tuple is inserted in $r$ we check if it is present in $s$, and if so we add it to $v$. If a tuple is deleted from $r$, we delete it from the intersection if it is present. The other set operations, *union* and *set difference*, are handled in a similar fashion; we leave details to you.

Outer joins are handled in much the same way as joins, but with some extra work. In the case of deletion from $r$ we have to handle tuples in $s$ that no longer match any tuple in $r$. In the case of insertion to $r$, we have to handle tuples in $s$ that did not match any tuple in $r$. Again we leave details to you.



#### 14.5.2.5 Handling Expressions

So far we have seen how to update incrementally the result of a single operation. To handle an entire expression, we can derive expressions for computing the incremental change to the result of each subexpression, starting from the smallest subexpressions.

For example, suppose we wish to incrementally update a materialized view $E_1 \bowtie E_2$ when a set of tuples $i_r$ is inserted into relation $r$. Let us assume $r$ is used in $E_1$ alone. Suppose the set of tuples to be inserted into $E_1$ is given by expression $D_1$. Then the expression $D_1 \bowtie E_2$ gives the set of tuples to be inserted into $E_1 \bowtie E_2$.

See the bibliographical notes for further details on incremental view maintenance with expressions.

### 14.5.3 Query Optimization and Materialized Views

Query optimization can be performed by treating materialized views just like regular relations. However, materialized views offer further opportunities for optimization:

- Rewriting queries to use materialized views:

  Suppose a materialized view $v = r \bowtie s$ is available, and a user submits a query $r \bowtie s \bowtie t$. Rewriting the query as $v \bowtie t$ may provide a more efficient query plan than optimizing the query as submitted. Thus, it is the job of the

be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction. This task may be facilitated by automatic testing of integrity constraints, as we discussed in Chapter 6.

- **Atomicity**: Suppose that, just before the execution of transaction $T_i$ the values of accounts $A$ and $B$ are \$1000 and \$2000, respectively. Now suppose that, during the execution of transaction $T_i$, a failure occurs that prevents $T_i$ from completing its execution successfully. Examples of such failures include power failures, hardware failures, and software errors. Further, suppose that the failure happened after the write($A$) operation but before the write($B$) operation. In this case, the values of accounts $A$ and $B$ reflected in the database are \$950 and \$2000. The system destroyed \$50 as a result of this failure. In particular, we note that the sum $A + B$ is no longer preserved.

  Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction $T_i$ is executed to completion, there exists a point at which the value of account $A$ is \$950 and the value of account $B$ is \$2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account $A$ is \$950, and the value of account $B$ is \$2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

  The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write, and, if the transaction does not complete its execution, the database system restores the old values to make it appear as though the transaction never executed. We discuss these ideas further in Section 15.2. Ensuring atomicity is the responsibility of the database system itself; specifically, it is handled by a component called the **transaction-management component**, which we describe in detail in Chapter 17.

- **Durability:** Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds.

  The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either

1. The updates carried out by the transaction have been written to disk before the transaction completes.
2. Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

Ensuring durability is the responsibility of a component of the database system called the **recovery-management component**. The transaction-management component and the recovery-management component are closely related, and we describe them in Chapter 17.

- **Isolation:** Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

  For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from $A$ to $B$ is executing, with the deducted total written to $A$ and the increased total yet to be written to $B$. If a second concurrently running transaction reads $A$ and $B$ at this intermediate point and computes $A + B$, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on $A$ and $B$ based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

  A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. However, concurrent execution of transactions provides significant performance benefits, as we shall see in Section 15.4. Other solutions have therefore been developed; they allow multiple transactions to execute concurrently.

  We discuss the problems caused by concurrently executing transactions in Section 15.4. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order. We shall discuss the principles of isolation further in Section 15.5. Ensuring the isolation property is the responsibility of a component of the database system called the **concurrency-control component**, which we discuss later, in Chapter 16.



## 15.2 Transaction State

In the absence of failures, all transactions complete successfully. However, as we noted earlier, a transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Thus, any changes that

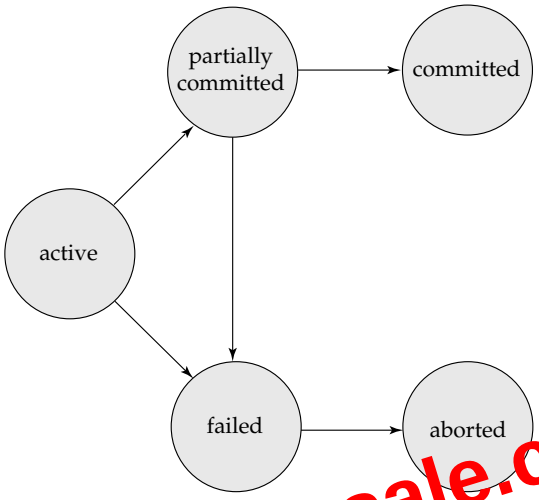



**Figure 15.1** State diagram of a transaction.



- It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
- It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

We must be cautious when dealing with **observable external writes**, such as writes to a terminal or printer. Once such a write has occurred, it cannot be erased, since it may have been seen external to the database system. Most systems allow such writes to take place only after the transaction has entered the committed state. One way to implement such a scheme is for the database system to store any value associated with such external writes temporarily in nonvolatile storage, and to perform the actual writes only after the transaction enters the committed state. If the system should fail after the transaction has entered the committed state, but before it could complete the external writes, the database system will carry out the external writes (using the data in nonvolatile storage) when the system is restarted.

Handling external writes can be more complicated in some situations. For example suppose the external action is that of dispensing cash at an automated teller machine, and the system fails just before the cash is actually dispensed (we assume that cash can be dispensed atomically). It makes no sense to dispense cash when the system is restarted, since the user may have left the machine. In such a case a compensating transaction, such as depositing the cash back in the users account, needs to be executed when the system is restarted.

## 15.6 Recoverability

So far, we have studied what schedules are acceptable from the viewpoint of consistency of the database, assuming implicitly that there are no transaction failures. We now address the effect of transaction failures during concurrent execution.

If a transaction $T_i$ fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction $T_j$ that is dependent on $T_i$ (that is, $T_j$ has read data written by $T_i$) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system.

In the following two subsections, we address the issue of what schedules are acceptable from the viewpoint of recovery from transaction failure. We describe in Chapter 16 how to ensure that only such acceptable schedules are generated.



### 15.6.1 Recoverable Schedules

Consider schedule 11 in Figure 15.13, in which $T_9$ is a transaction that performs only one instruction: read($A$). Suppose that the system allows $T_9$ to commit immediately after executing the read($A$) instruction. Thus $T_9$ commits before $T_8$ does. Now suppose that $T_8$ fails before it commits. Since $T_9$ has read the value of data item $A$ written by $T_8$, we must abort $T_9$ to ensure transaction atomicity. However, $T_9$ has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of $T_8$.

Schedule 11, with the commit happening immediately after the read($A$) instruction, is an example of a *nonrecoverable* schedule, which should not be allowed. Most database system require that all schedules be *recoverable*. A **recoverable schedule** is one where, for each pair of transactions $T_i$ and $T_j$ such that $T_j$ reads a data item previously written by $T_i$, the commit operation of $T_i$ appears before the commit operation of $T_j$.

### 15.6.2 Cascadeless Schedules

Even if a schedule is recoverable, to recover correctly from the failure of a transaction $T_i$, we may have to roll back several transactions. Such situations occur if transactions have read data written by $T_i$. As an illustration, consider the partial schedule

| $T_8$ | $T_9$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| read($B$) | |

**Figure 15.13** Schedule 11.

The goal of concurrency-control schemes is to provide a high degree of concurrency, while ensuring that all schedules that can be generated are conflict or view serializable, and are cascadeless.

We study a number of concurrency-control schemes in Chapter 16. The schemes have different trade-offs in terms of the amount of concurrency they allow and the amount of overhead that they incur. Some of them allow only conflict serializable schedules to be generated; others allow certain view-serializable schedules that are not conflict-serializable to be generated.

## 15.8 Transaction Definition in SQL

A data-manipulation language must include a construct for specifying the set of actions that constitute a transaction.

The SQL standard specifies that a transaction begins implicitly. Transactions are ended by one of these SQL statements:

- **Commit work** commits the current transaction and begins a new one.
- **Rollback work** causes the current transaction to abort.

The keyword **work** is optional in both the statements. If a program terminates without either of these commands, the updates are either committed or rolled back—which of the two happens is not specified by the standard and depends on the implementation.



The standard also specifies that the system must ensure both serializability and freedom from cascading rollback. The definition of serializability used by the standard is that a schedule must have the *same effect* as would some serial schedule. Thus, conflict and view serializability are both acceptable.

The SQL-92 standard also allows a transaction to specify that it may be executed in a manner that causes it to become nonserializable with respect to other transactions. We study such weaker levels of consistency in Section 16.8.

## 15.9 Testing for Serializability

When designing concurrency control schemes, we must show that schedules generated by the scheme are serializable. To do that, we must first understand how to determine, given a particular schedule $S$, whether the schedule is serializable.

We now present a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule $S$. We construct a directed graph, called a **precedence graph**, from $S$. This graph consists of a pair $G = (V, E)$, where $V$ is a set of vertices and $E$ is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:

1. $T_i$ executes write($Q$) before $T_j$ executes read($Q$).
2. $T_i$ executes read($Q$) before $T_j$ executes write($Q$).
3. $T_i$ executes write($Q$) before $T_j$ executes write($Q$).

$T_2$: lock-S($A$);
read($A$);
unlock($A$);
lock-S($B$);
read($B$);
unlock($B$);
display($A + B$).

**Figure 16.3** Transaction $T_2$.

Transaction $T_i$ may unlock a data item that it had locked at some earlier point. Note that a transaction must hold a lock on a data item as long as it accesses that item. Moreover, for a transaction to unlock a data item immediately after its final access of that data item is not always desirable, since serializability may not be ensured.

As an illustration, consider again the simplified banking system that we introduced in Chapter 15. Let $A$ and $B$ be two accounts that are accessed by transactions $T_1$ and $T_2$. Transaction $T_1$ transfers \$50 from account $B$ to account $A$ (Figure 16.2). Transaction $T_2$ displays the total amount of money in accounts $A$ and $B$—that is, the sum $A + B$ (Figure 16.3).



| $T_1$ | $T_2$ | concurrency-control manager |
|---|---|---|
| lock-X($B$) | | |
| | | grant-X($B$, $T_1$) |
| read($B$) | | |
| $B := B - 50$ | | |
| write($B$) | | |
| unlock($B$) | | |
| | lock-S($A$) | |
| | | grant-S($A$, $T_2$) |
| | read($A$) | |
| | unlock($A$) | |
| | lock-S($B$) | |
| | | grant-S($B$, $T_2$) |
| | read($B$) | |
| | unlock($B$) | |
| | display($A + B$) | |
| lock-X($A$) | | |
| | | grant-X($A$, $T_2$) |
| read($A$) | | |
| $A := A + 50$ | | |
| write($A$) | | |
| unlock($A$) | | |

**Figure 16.4** Schedule 1.

**b.** If TS($T_i$) $\geq$ W-timestamp($Q$), then the read operation is executed, and R-timestamp($Q$) is set to the maximum of R-timestamp($Q$) and TS($T_i$).

**2.** Suppose that transaction $T_i$ issues write($Q$).

**a.** If TS($T_i$) $<$ R-timestamp($Q$), then the value of $Q$ that $T_i$ is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls $T_i$ back.

**b.** If TS($T_i$) $<$ W-timestamp($Q$), then $T_i$ is attempting to write an obsolete value of $Q$. Hence, the system rejects this write operation and rolls $T_i$ back.

**c.** Otherwise, the system executes the write operation and sets W-timestamp($Q$) to TS($T_i$).

If a transaction $T_i$ is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

To illustrate this protocol, we consider transactions $T_{14}$ and $T_{15}$. Transaction $T_{14}$ displays the contents of accounts $A$ and $B$:

$T_{14}$: read($B$);
read($A$);
display($A + B$).



Transaction $T_{15}$ transfers \$50 from account $A$ to account $B$, and then displays the contents of both:

$T_{15}$: read($B$);
$B := B - 50$;
write($B$);
read($A$);
$A := A + 50$;
write($A$);
display($A + B$).

In presenting schedules under the timestamp protocol, we shall assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3 of Figure 16.13, TS($T_{14}$) $<$ TS($T_{15}$), and the schedule is possible under the timestamp protocol.

We note that the preceding execution can also be produced by the two-phase locking protocol. There are, however, schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa (see Exercise 16.20).

The timestamp-ordering protocol ensures conflict serializability. This is because conflicting operations are processed in timestamp order.

The protocol ensures freedom from deadlock, since no transaction ever waits. However, there is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction. If

tive scheme that imposes less overhead. A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for **monitoring** the system.

We assume that each transaction $T_i$ executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order,

1. **Read phase**. During this phase, the system executes transaction $T_i$. It reads the values of the various data items and stores them in variables local to $T_i$. It performs all write operations on temporary local variables, without updates of the actual database.
2. **Validation phase**. Transaction $T_i$ performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serializability.
3. **Write phase**. If transaction $T_i$ succeeds in validation (step 2), then the system applies the actual updates to the database. Otherwise, the system rolls back $T_i$.



Each transaction must go through the three phases in the order shown. However, all three phases of concurrently executing transactions can be interleaved.

To perform the validation test, we need to know when the various phases of transactions $T_i$ took place. We shall, therefore, associate three different timestamps with transaction $T_i$:

1. **Start**($T_i$), the time when $T_i$ started its execution.
2. **Validation**($T_i$), the time when $T_i$ finished its read phase and started its validation phase.
3. **Finish**($T_i$), the time when $T_i$ finished its write phase.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation($T_i$). Thus, the value TS($T_i$) = Validation($T_i$) and, if TS($T_j$) < TS($T_k$), then any produced schedule must be equivalent to a serial schedule in which transaction $T_j$ appears before transaction $T_k$. The reason we have chosen Validation($T_i$), rather than Start($T_i$), as the timestamp of transaction $T_i$ is that we can expect faster response time provided that conflict rates among transactions are indeed low.

The **validation test** for transaction $T_j$ requires that, for all transactions $T_i$ with TS($T_i$) < TS($T_j$), one of the following two conditions must hold:

1. Finish($T_i$) < Start($T_j$). Since $T_i$ completes its execution before $T_j$ started, the serializability order is indeed maintained.
2. The set of data items written by $T_i$ does not intersect with the set of data items read by $T_j$, and $T_i$ completes its write phase before $T_j$ starts its validation phase (Start($T_j$) < Finish($T_i$) < Validation($T_j$)). This condition ensures that

Observe that the multiple-granularity protocol requires that locks be acquired in *top-down* (root-to-leaf) order, whereas locks must be released in *bottom-up* (leaf-to-root) order.

As an illustration of the protocol, consider the tree of Figure 16.16 and these transactions:

- Suppose that transaction $T_{18}$ reads record $r_{a_2}$ in file $F_a$. Then, $T_{18}$ needs to lock the database, area $A_1$, and $F_a$ in IS mode (and in that order), and finally to lock $r_{a_2}$ in S mode.
- Suppose that transaction $T_{19}$ modifies record $r_{a_9}$ in file $F_a$. Then, $T_{19}$ needs to lock the database, area $A_1$, and file $F_a$ in IX mode, and finally to lock $r_{a_9}$ in X mode.
- Suppose that transaction $T_{20}$ reads all the records in file $F_a$. Then, $T_{20}$ needs to lock the database and area $A_1$ (in that order) in IS mode, and finally to lock $F_a$ in S mode.
- Suppose that transaction $T_{21}$ reads the entire database. It can do so after locking the database in S mode.

We note that transactions $T_{18}$, $T_{20}$, and $T_{21}$ can access the database concurrently. Transaction $T_{19}$ can execute concurrently with $T_{18}$, but not with either $T_{20}$ or $T_{21}$.

This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of

- Short transactions that access only a few data items
- Long transactions that produce reports from an entire file or set of files

There is a similar locking protocol that is applicable to database systems in which data granularities are organized in the form of a directed acyclic graph. See the bibliographical notes for additional references. Deadlock is possible in the protocol that we have, as it is in the two-phase locking protocol. There are techniques to reduce deadlock frequency in the multiple-granularity protocol, and also to eliminate deadlock entirely. These techniques are referenced in the bibliographical notes.



## 16.5 Multiversion Schemes

The concurrency-control schemes discussed thus far ensure serializability by either delaying an operation or aborting the transaction that issued the operation. For example, a read operation may be delayed because the appropriate value has not been written yet; or it may be rejected (that is, the issuing transaction must be aborted) because the value that it was supposed to read has already been overwritten. These difficulties could be avoided if old copies of each data item were kept in a system.

In **multiversion concurrency control** schemes, each write($Q$) operation creates a new **version** of $Q$. When a transaction issues a read($Q$) operation, the concurrency-control manager selects one of the versions of $Q$ to be read. The concurrency-control scheme must ensure that the version to be read is selected in a manner that ensures

- If $T_{29}$ uses the tuple newly inserted by $T_{30}$ in computing **sum**(*balance*), then $T_{29}$ read a value written by $T_{30}$. Thus, in a serial schedule equivalent to $S$, $T_{30}$ must come before $T_{29}$.
- If $T_{29}$ does not use the tuple newly inserted by $T_{30}$ in computing **sum**(*balance*), then in a serial schedule equivalent to $S$, $T_{29}$ must come before $T_{30}$.

The second of these two cases is curious. $T_{29}$ and $T_{30}$ do not access any tuple in common, yet they conflict with each other! In effect, $T_{29}$ and $T_{30}$ conflict on a phantom tuple. If concurrency control is performed at the tuple granularity, this conflict would go undetected. This problem is called the **phantom phenomenon**.

To prevent the phantom phenomenon, we allow $T_{29}$ to prevent other transactions from creating new tuples in the *account* relation with *branch-name* = "Perryridge."

To find all *account* tuples with *branch-name* = "Perryridge", $T_{29}$ must search either the whole *account* relation, or at least an index on the relation. Up to now, we have assumed implicitly that the only data items accessed by a transaction are tuples. However, $T_{29}$ is an example of a transaction that reads information about what tuples are in a relation, and $T_{30}$ is an example of a transaction that updates that information.

Clearly, it is not sufficient merely to lock the tuples that are accessed; the information used to find the tuples that are accessed by the transaction must also be locked.

The simplest solution to this problem is to associate a data item with the relation; the data item represents the information used to find the tuples in the relation. Transactions, such as $T_{29}$, that read the information about what tuples are in a relation would then have to lock the data item corresponding to the relation in shared mode. Transactions, such as $T_{30}$, that update the information about what tuples are in a relation would have to lock the data item in exclusive mode. Thus, $T_{29}$ and $T_{30}$ would conflict on a real data item, rather than on a phantom.

Do not confuse the locking of an entire relation, as in multiple granularity locking, with the locking of the data item corresponding to the relation. By locking the data item, a transaction only prevents other transactions from updating information about what tuples are in the relation. Locking is still required on tuples. A transaction that directly accesses a tuple can be granted a lock on the tuples even when another transaction has an exclusive lock on the data item corresponding to the relation itself.

The major disadvantage of locking a data item corresponding to the relation is the low degree of concurrency— two transactions that insert different tuples into a relation are prevented from executing concurrently.

A better solution is the **index-locking** technique. Any transaction that inserts a tuple into a relation must insert information into every index maintained on the relation. We eliminate the phantom phenomenon by imposing a locking protocol for indices. For simplicity we shall only consider $B^+$-tree indices.

As we saw in Chapter 12, every search-key value is associated with an index leaf node. A query will usually use one or more indices to access a relation. An insert must insert the new tuple in all indices on the relation. In our example, we assume that there is an index on *account* for *branch-name*. Then, $T_{30}$ must modify the leaf containing the key Perryridge. If $T_{29}$ reads the same leaf node to locate all tuples pertaining to the Perryridge branch, then $T_{29}$ and $T_{30}$ conflict on that leaf node.

these transactions were to execute in a serializable fashion, they could interfere with other transactions, causing the others' execution to be delayed.

The levels of consistency specified by SQL-92 are as follows:

- **Serializable** is the default.
- **Repeatable read** allows only committed records to be read, and further requires that, between two reads of a record by a transaction, no other transaction is allowed to update the record. However, the transaction may not be serializable with respect to other transactions. For instance, when it is searching for records satisfying some conditions, a transaction may find some of the records inserted by a committed transaction, but may not find others.
- **Read committed** allows only committed records to be read, but does not require even repeatable reads. For instance, between two reads of a record by the transaction, the records may have been updated by other committed transactions. This is basically the same as degree-two consistency; most systems supporting this level of consistency would actually implement cursor stability, which is a special case of degree-two consistency.
- **Read uncommitted** allows even uncommitted records to be read. It is the lowest level of consistency allowed by SQL-92.



## 16.9 Concurrency in Index Structures∗∗

It is possible to treat access to index structures like any other database structure, and to apply the concurrency-control techniques discussed earlier. However, since indices are accessed frequently, they would become a point of great lock contention, leading to a low degree of concurrency. Luckily, indices do not have to be treated like other database structures. It is perfectly acceptable for a transaction to perform a lookup on an index twice, and to find that the structure of the index has changed in between, as long as the index lookup returns the correct set of tuples. Thus, it is acceptable to have nonserializable concurrent access to an index, as long as the accuracy of the index is maintained.

We outline two techniques for managing concurrent access to $B^+$-trees. The bibliographical notes reference other techniques for $B^+$-trees, as well as techniques for other index structures.

The techniques that we present for concurrency control on $B^+$-trees are based on locking, but neither two-phase locking nor the tree protocol is employed. The algorithms for lookup, insertion, and deletion are those used in Chapter 12, with only minor modifications.

The first technique is called the **crabbing protocol**:

- When searching for a key value, the crabbing protocol first locks the root node in shared mode. When traversing down the tree, it acquires a shared lock on the child node to be traversed further. After acquiring the lock on the child node, it releases the lock on the parent node. It repeats this process until it reaches a leaf node.

or deletion. It locks leaf nodes affected by insertion or deletion following the two-phase locking protocol, as Section 16.7.3 describes, to avoid the phantom phenomenon.

- **Split**. If the transaction splits a node, it creates a new node according to the algorithm of Section 12.3 and makes it the right sibling of the original node. The right-sibling pointers of both the original node and the new node are set. Following this, the transaction releases the exclusive lock on the original node and requests an exclusive lock on the parent, so that it can insert a pointer to the new node.
- **Coalescence**. If a node has too few search-key values after a deletion, the node with which it will be coalesced must be locked in exclusive mode. Once the transaction has coalesced these two nodes, it requests an exclusive lock on the parent so that the deleted node can be removed. At this point, the transaction releases the locks on the coalesced nodes. Unless the parent node must be coalesced also, its lock is released.



Observe this important fact: An insertion or deletion may lock a node, unlock it, and subsequently relock it. Furthermore, a lookup that runs concurrently with a split or coalescence operation may find that the desired search key has been moved to the right-hand node by the split or coalescence operation.

As an illustration, consider the B$^+$-tree in Figure 16.21. Assume that there are two concurrent operations on this B$^+$-tree:

1. Insert "Clearview"
2. Look up "Downtown"

Let us assume that the insertion operation begins first. It does a lookup on "Clearview," and finds that the node into which "Clearview" should be inserted is full. It therefore converts its shared lock on the node to exclusive mode, and creates a new node. The original node now contains the search-key values "Brighton" and "Clearview." The new node contains the search-key value "Downtown."

Now assume that a context switch occurs that results in control passing to the lookup operation. This lookup operation accesses the root, and follows the pointer

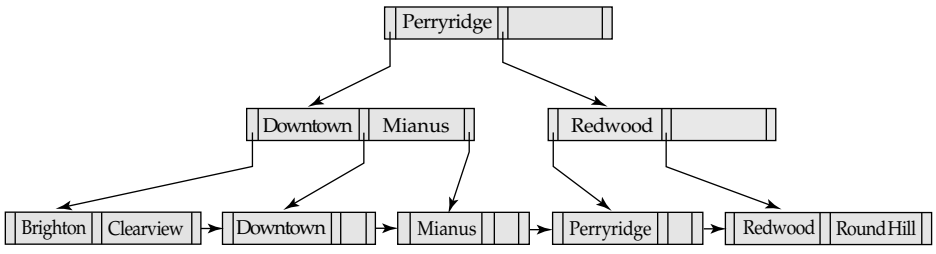



**Figure 16.21** B$^+$-tree for *account* file with $n = 3$.

## 16.10 Summary

- When several transactions execute concurrently in the database, the consistency of data may no longer be preserved. It is necessary for the system to control the interaction among the concurrent transactions, and this control is achieved through one of a variety of mechanisms called *concurrency-control* schemes.
- To ensure serializability, we can use various concurrency-control schemes. All these schemes either delay an operation or abort the transaction that issued the operation. The most common ones are locking protocols, timestamp-ordering schemes, validation techniques, and multiversion schemes.
- A locking protocol is a set of rules that state when a transaction may lock and unlock each of the data items in the database.
- The two-phase locking protocol allows a transaction to lock a new data item only if that transaction has not yet unlocked any data item. The protocol ensures serializability, but not deadlock freedom. In the absence of information concerning the manner in which data items are accessed, the two-phase locking protocol is both necessary and sufficient for ensuring serializability.
- The strict two-phase locking protocol permits release of exclusive locks only at the end of transaction, in order to ensure recoverability and cascadelessness of the resulting schedules. The rigorous two-phase locking protocol releases all locks only at the end of the transaction.
- A timestamp-ordering scheme ensures serializability by selecting an ordering in advance between every pair of transactions. A unique fixed timestamp is associated with each transaction in the system. The timestamps of the transactions determine the serializability order. Thus, if the timestamp of transaction $T_i$ is smaller than the timestamp of transaction $T_j$, then the scheme ensures that the produced schedule is equivalent to a serial schedule in which transaction $T_i$ appears before transaction $T_j$. It does so by rolling back a transaction whenever such an order is violated.
- A validation scheme is an appropriate concurrency-control method in cases where a majority of transactions are read-only transactions, and thus the rate of conflicts among these transactions is low. A unique fixed timestamp is associated with each transaction in the system. The serializability order is determined by the timestamp of the transaction. A transaction in this scheme is never delayed. It must, however, pass a validation test to complete. If it does not pass the validation test, the system rolls it back to its initial state.
- There are circumstances where it would be advantageous to group several data items, and to treat them as one aggregate data item for purposes of working, resulting in multiple levels of granularity. We allow data items of various sizes, and define a hierarchy of data items, where the small items are nested within larger ones. Such a hierarchy can be represented graphically as a tree.

- Special concurrency-control techniques can be developed for special data structures. Often, special techniques are applied in $B^+$-trees to allow greater concurrency. These techniques allow nonserializable access to the $B^+$-tree, but they ensure that the $B^+$-tree structure is correct, and ensure that accesses to the database itself are serializable.

## Review Terms

- Concurrency control
- Lock types
  - Shared-mode (S) lock
  - Exclusive-mode (X) lock
- Lock
  - Compatibility
  - Request
  - Wait
  - Grant
- Deadlock
- Starvation
- Locking protocol
- Legal schedule
- Two-phase locking protocol
  - Growing phase
  - Shrinking phase
  - Lock point
  - Strict two-phase locking
  - Rigorous two-phase locking
- Lock conversion
  - Upgrade
  - Downgrade
- Graph-based protocols
  - Tree protocol
  - Commit dependency
- Timestamp-based protocols
- Timestamp
  - System clock
  - Logical counter
  - W-timestamp($Q$)
  - R-timestamp($Q$)
- Timestamp-ordering protocol
  - Thomas' write rule
- Validation-based protocols
  - Read phase
  - Validation phase
  - Write phase
  - Validation test
- Multiple granularity
  - Explicit locks
  - Implicit locks
  - Intention locks
- Intention lock modes
  - Intention-shared (IS)
  - Intention-exclusive (IX)
  - Shared and intention-exclusive (SIX)
- Multiple-granularity locking protocol
- Multiversion concurrency control
- Versions
- Multiversion timestamp ordering
- Multiversion two-phase locking
  - Read-only transactions
  - Update transactions
- Deadlock handling
  - Prevention
  - Detection
  - Recovery
- Deadlock prevention
  - Ordered locking
  - Preemption of locks
  - Wait–die scheme
  - Wound–wait scheme
  - Timeout-based schemes

**16.30** Suppose that we use the tree protocol of Section 16.1.5 to manage concurrent access to a $B^+$-tree. Since a split may occur on an insert that affects the root, it appears that an insert operation cannot release any locks until it has completed the entire operation. Under what circumstances is it possible to release a lock earlier?

**16.31** Give example schedules to show that if any of lookup, insert or delete do not lock the next key value, the phantom phenomemon could go undetected.

## Bibliographical Notes

Gray and Reuter [1993] provides detailed textbook coverage of transaction-processing concepts, including concurrency control concepts and implementation details. Bernstein and Newcomer [1997] provides textbook coverage of various aspects of transaction processing including concurrency control.

Early textbook discussions of concurrency control and recovery included Papadimitriou [1986] and Bernstein et al. [1987]. An early survey paper on implementation issues in concurrency control and recovery is presented by Gray [1978].

The two-phase locking protocol was introduced by Eswaran et al. [1976]. The tree-locking protocol is from Silberschatz and Kedem [1980]. Other non-two-phase locking protocols that operate on more general graphs are described in Yannakakis et al. [1979], Kedem and Silberschatz [1983], and Buckley and Silberschatz [1985]. General discussions concerning locking protocols are offered by Lien and Weinberger [1978], Yannakakis et al. [1979], Yannakakis [1981], and Papadimitriou [1982]. Korth [1983] explores various lock modes that can be obtained from the basic shared and exclusive lock modes.

Exercise 16.6 is from Buckley and Silberschatz [1984]. Exercise 16.8 is from Kedem and Silberschatz [1983]. Exercise 16.9 is from Kedem and Silberschatz [1979]. Exercise 16.10 is from Yannakakis et al. [1979]. Exercise 16.13 is from Korth [1983].

The timestamp-based concurrency-control scheme is from Reed [1983]. An exposition of various timestamp-based concurrency-control algorithms is presented by Bernstein and Goodman [1980]. A timestamp algorithm that does not require any rollback to ensure serializability is presented by Buckley and Silberschatz [1983]. The validation concurrency-control scheme is from Kung and Robinson [1981].

The locking protocol for multiple-granularity data items is from Gray et al. [1975]. A detailed description is presented by Gray et al. [1976]. The effects of locking granularity are discussed by Ries and Stonebraker [1977]. Korth [1983] formalizes multiple-granularity locking for an arbitrary collection of lock modes (allowing for more semantics than simply read and write). This approach includes a class of lock modes called *update* modes to deal with lock conversion. Carey [1983] extends the multiple-granularity idea to timestamp-based concurrency control. An extension of the protocol to ensure deadlock freedom is presented by Korth [1982]. Multiple-granularity locking for object-oriented database systems is discussed in Lee and Liou [1996].

Discussions concerning multiversion concurrency control are offered by Bernstein et al. [1983]. A multiversion tree-locking algorithm appears in Silberschatz [1982].

CHAPTER 17

# Recovery System



A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage. In any failure, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions, introduced in Chapter 15, are preserved. An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide **high availability**; that is, it must minimize the time for which the database is not usable after a crash.

## 17.1 Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. The simplest type of failure is one that does not result in the loss of information in the system. The failures that are more difficult to deal with are those that result in loss of information. In this chapter, we shall consider only the following types of failure:

- **Transaction failure.** There are two types of errors that may cause a transaction to fail:
  - **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
  - **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be reexecuted at a later time.
- **System crash**. There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile

storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.

The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the **fail-stop assumption**. Well-designed systems have numerous internal checks, at the hardware and the software level, that bring the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one.

- **Disk failure**. A disk block loses its content as a result of either a head crash or failure during a data transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as tapes, are used to recover from the failure.

To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data. Next, we must consider how these failure modes affect the contents of the database. We can then propose algorithms to ensure database consistency and transaction atomicity despite failures. These algorithms, known as recovery algorithms, have two parts:

1. Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.
2. Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.



## 17.2 Storage Structure

As we saw in Chapter 11, the various data items in the database may be stored and accessed in a number of different storage media. To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of these storage media and their access methods.

### 17.2.1 Storage Types

In Chapter 11 we saw that storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as volatile storage or nonvolatile storage. We review these terms, and introduce another class of storage, called stable storage.

- **Volatile storage**. Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.
- **Nonvolatile storage**. Information residing in nonvolatile storage survives system crashes. Examples of such storage are disk and magnetic tapes. Disks are used for online storage, whereas tapes are used for archival storage. Both,

however, are subject to failure (for example, head crash), which may result in loss of information. At the current state of technology, nonvolatile storage is slower than volatile storage by several orders of magnitude. This is because disk and tape devices are electromechanical, rather than based entirely on chips, as is volatile storage. In database systems, disks are used for most nonvolatile storage. Other nonvolatile media are normally used only for backup data. Flash storage (see Section 11.1), though nonvolatile, has insufficient capacity for most database systems.

- **Stable storage**. Information residing in stable storage is *never* lost (*never* should be taken with a grain of salt, since theoretically *never* cannot be guaranteed—for example, it is possible, although extremely unlikely, that a black hole may envelop the earth and permanently destroy all data!). Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely. Section 17.2.2 discusses stable-storage implementation.

The distinctions among the various storage types are often less clear in practice than in our presentation. Certain systems provide battery backup, so that some main memory can survive system crashes and power failures. Alternative forms of nonvolatile storage, such as optical media, provide an even higher degree of reliability than do disks.



### 17.2.2 Stable-Storage Implementation

To implement stable storage, we need to replicate the needed information in several nonvolatile storage media (usually disk) with independent failure modes, and to update the information in a controlled manner to ensure that failure during data transfer does not damage the needed information.

Recall (from Chapter 11) that RAID systems guarantee that the failure of a single disk (even during data transfer) will not result in loss of data. The simplest and fastest form of RAID is the mirrored disk, which keeps two copies of each block, on separate disks. Other forms of RAID offer lower costs, but at the expense of lower performance.

RAID systems, however, cannot guard against data loss due to disasters such as fires or flooding. Many systems store archival backups of tapes off-site to guard against such disasters. However, since tapes cannot be carried off-site continually, updates since the most recent time that tapes were carried off-site could be lost in such a disaster. More secure systems keep a copy of each block of stable storage at a remote site, writing it out over a computer network, in addition to storing the block on a local disk system. Since the blocks are output to a remote system as and when they are output to local storage, once an output operation is complete, the output is not lost, even in the event of a disaster such as a fire or flood. We study such *remote backup* systems in Section 17.10.

In the remainder of this section, we discuss how storage media can be protected from failure during data transfer. Block transfer between memory and disk storage can result in

- **Successful completion**. The transferred information arrived safely at its destination.
- **Partial failure**. A failure occurred in the midst of transfer, and the destination block has incorrect information.
- **Total failure**. The failure occurred sufficiently early during the transfer that the destination block remains intact.

We require that, if a **data-transfer failure** occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do so, the system must maintain two physical blocks for each logical database block; in the case of mirrored disks, both blocks are at the same location; in the case of remote backup, one of the blocks is local, whereas the other is at a remote site. An output operation is executed as follows:

1. Write the information onto the first physical block.
2. When the first write completed successfully, write the same information onto the second physical block.
3. The output is completed only after the second write completes successfully.



During recovery, the system examines each pair of physical blocks. If both are the same and no detectable error exists, then no further actions are necessary. (Recall that errors in a disk block, such as a partial write to the block, are detected by storing a checksum with each block.) If the system detects an error in one block, then it replaces its content with the content of the other block. If both blocks contain no detectable error, but they differ in content, then the system replaces the content of the first block with the value of the second. This recovery procedure ensures that a write to stable storage either succeeds completely (that is, updates all copies) or results in no change.

The requirement of comparing every corresponding pair of blocks during recovery is expensive to meet. We can reduce the cost greatly by keeping track of block writes that are in progress, using a small amount of nonvolatile RAM. On recovery, only blocks for which writes were in progress need to be compared.

The protocols for writing out a block to a remote site are similar to the protocols for writing blocks to a mirrored disk system, which we examined in Chapter 11, and particularly in Exercise 11.4.

We can extend this procedure easily to allow the use of an arbitrarily large number of copies of each block of stable storage. Although a large number of copies reduces the probability of a failure to even lower than two copies do, it is usually reasonable to simulate stable storage with only two copies.

### 17.2.3 Data Access

As we saw in Chapter 11, the database system resides permanently on nonvolatile storage (usually disks), and is partitioned into fixed-length storage units called **blocks**. Blocks are the units of data transfer to and from disk, and may contain several data

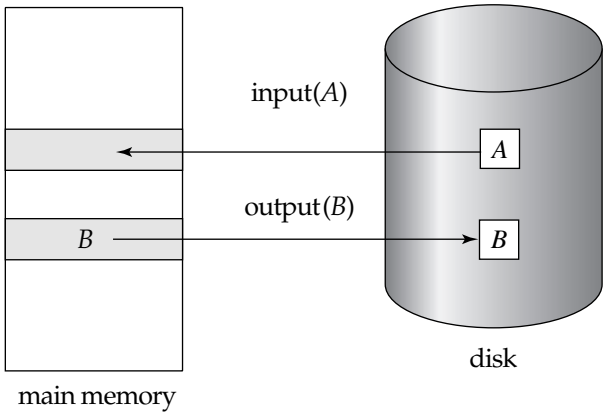



**Figure 17.1** Block storage operations.



items. We shall assume that no data item spans two or more blocks. This assumption is realistic for most data-processing applications, such as our banking example.

Transactions input information from the disk to main memory, and then output the information back onto the disk. The input and output operations are done in block units. The blocks residing on the disk are referred to as **physical blocks**; the blocks residing temporarily in main memory are referred to as **buffer blocks**. The area of memory where blocks reside temporarily is called the **disk buffer**.

Block movements between disk and main memory are initiated through the following two operations:

1. input($B$) transfers the physical block $B$ to main memory.
2. output($B$) transfers the buffer block $B$ to the disk, and replaces the appropriate physical block there.

Figure 17.1 illustrates this scheme.

Each transaction $T_i$ has a private work area in which copies of all the data items accessed and updated by $T_i$ are kept. The system creates this work area when the transaction is initiated; the system removes it when the transaction either commits or aborts. Each data item $X$ kept in the work area of transaction $T_i$ is denoted by $x_i$. Transaction $T_i$ interacts with the database system by transferring data to and from its work area to the system buffer. We transfer data by these two operations:

1. read($X$) assigns the value of data item $X$ to the local variable $x_i$. It executes this operation as follows:
   a. If block $B_X$ on which $X$ resides is not in main memory, it issues input($B_X$).
   b. It assigns to $x_i$ the value of $X$ from the buffer block.
2. write($X$) assigns the value of local variable $x_i$ to data item $X$ in the buffer block. It executes this operation as follows:
   a. If block $B_X$ on which $X$ resides is not in main memory, it issues input($B_X$).
   b. It assigns the value of $x_i$ to $X$ in buffer $B_X$.

Note that both operations may require the transfer of a block from disk to main memory. They do not, however, specifically require the transfer of a block from main memory to disk.

A buffer block is eventually written out to the disk either because the buffer manager needs the memory space for other purposes or because the database system wishes to reflect the change to $B$ on the disk. We shall say that the database system performs a **force-output** of buffer $B$ if it issues an output($B$).

When a transaction needs to access a data item $X$ for the first time, it must execute read($X$). The system then performs all updates to $X$ on $x_i$. After the transaction accesses $X$ for the final time, it must execute write($X$) to reflect the change to $X$ in the database itself.

The output($B_X$) operation for the buffer block $B_X$ on which $X$ resides does not need to take effect immediately after write($X$) is executed, since the block $B_X$ may contain other data items that are still being accessed. Thus, the actual output may take place later. Notice that, if the system crashes after the write($X$) operation was executed but before output($B_X$) was executed, the new value of $X$ is never written to disk and, thus, is lost.



## 17.3 Recovery and Atomicity

Consider again our simplified banking system and transaction $T_i$ that transfers \$50 from account $A$ to account $B$, with initial values of $A$ and $B$ being \$1000 and \$2000, respectively. Suppose that a system crash has occurred during the execution of $T_i$, after output($B_A$) has taken place, but before output($B_B$) was executed, where $B_A$ and $B_B$ denote the buffer blocks on which $A$ and $B$ reside. Since the memory contents were lost, we do not know the fate of the transaction; thus, we could invoke one of two possible recovery procedures:

- **Reexecute** $T_i$. This procedure will result in the value of $A$ becoming \$900, rather than \$950. Thus, the system enters an inconsistent state.
- **Do not reexecute** $T_i$. The current system state has values of \$950 and \$2000 for $A$ and $B$, respectively. Thus, the system enters an inconsistent state.

In either case, the database is left in an inconsistent state, and thus this simple recovery scheme does not work. The reason for this difficulty is that we have modified the database without having assurance that the transaction will indeed commit. Our goal is to perform either all or no database modifications made by $T_i$. However, if $T_i$ performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.

To achieve our goal of atomicity, we must first output information describing the modifications to stable storage, without modifying the database itself. As we shall see, this procedure will allow us to output all the modifications made by a committed transaction, despite failures. There are two ways to perform such outputs; we study them in Sections 17.4 and 17.5. In these two sections, we shall assume that

result of the redo operations, but all changes may not have been made. When the system comes up after the second crash, recovery proceeds exactly as in the preceding examples. For each commit record

$$<T_i \text{ commit}>$$

found in the log, the the system performs the operation redo($T_i$). In other words, it restarts the recovery actions from the beginning. Since redo writes values to the database independent of the values currently in the database, the result of a successful second attempt at redo is the same as though redo had succeeded the first time.

### 17.4.2 Immediate Database Modification

The **immediate-modification technique** allows database modifications to be output to the database while the transaction is still in the active state. Data modifications written by active transactions are called **uncommitted modifications**. In the event of a crash or a transaction failure, the system must use the old-value field of the log records described in Section 17.4.1 to restore the modified data items to the value they had prior to the start of the transaction. The undo operation, described next, accomplishes this restoration.

Before a transaction $T_i$ starts its execution, the system writes the record $<T_i$ start$>$ to the log. During its execution, any write($X$) operation by $T_i$ is *preceded* by the writing of the appropriate new update record to the log. When $T_i$ partially commits, the system writes the record $<T_i$ commit$>$ to the log.

Since the information in the log is used in reconstructing the state of the database, we cannot allow the actual update to the database to take place before the corresponding log record is written out to stable storage. We therefore require that, before execution of an output($B$) operation, the log records corresponding to $B$ be written onto stable storage. We shall return to this issue in Section 17.7.

As an illustration, let us reconsider our simplified banking system, with transactions $T_0$ and $T_1$ executed one after the other in the order $T_0$ followed by $T_1$. The portion of the log containing the relevant information concerning these two transactions appears in Figure 17.5.

Figure 17.6 shows one possible order in which the actual outputs took place in both the database system and the log as a result of the execution of $T_0$ and $T_1$. Notice that

<$T_0$ start>
<$T_0$, $A$, 1000, 950>
<$T_0$, $B$, 2000, 2050>
<$T_0$ commit>
<$T_1$ start>
<$T_1$, $C$, 700, 600>
<$T_1$ commit>

**Figure 17.5** Portion of the system log corresponding to $T_0$ and $T_1$.

item, it must acquire an exclusive lock on the block in which the data item resides. The lock can be released immediately after the update has been performed. Before a block is output, the system obtains an exclusive lock on the block, to ensure that no transaction is updating the block. It releases the lock once the block output has completed. Locks that are held for a short duration are often called **latches**. Latches are treated as distinct from locks used by the concurrency-control system. As a result, they may be released without regard to any locking protocol, such as two-phase locking, required by the concurrency-control system.

To illustrate the need for the write-ahead logging requirement, consider our banking example with transactions $T_0$ and $T_1$. Suppose that the state of the log is

$$<T_0 \text{ start}>$$
$$<T_0,\ A,\ 1000,\ 950>$$

and that transaction $T_0$ issues a read($B$). Assume that the block on which $B$ resides is not in main memory, and that main memory is full. Suppose that the block on which $A$ resides is chosen to be output to disk. If the system outputs this block to disk and then a crash occurs, the values in the database for accounts $A$, $B$, and $C$ are \$950, \$2000, and \$700, respectively. This database state is inconsistent. However, because of the WAL requirements, the log record

$$<T_0,\ A,\ 1000,\ 950>$$

must be output to stable storage prior to output of the block on which $A$ resides. The system can use the log record during recovery to bring the database back to a consistent state.



### 17.7.3 Operating System Role in Buffer Management

We can manage the database buffer by using one of two approaches:

1. The database system reserves part of main memory to serve as a buffer that it, rather than the operating system, manages. The database system manages data-block transfer in accordance with the requirements in Section 17.7.2.

   This approach has the drawback of limiting flexibility in the use of main memory. The buffer must be kept small enough that other applications have sufficient main memory available for their needs. However, even when the other applications are not running, the database will not be able to make use of all the available memory. Likewise, nondatabase applications may not use that part of main memory reserved for the database buffer, even if some of the pages in the database buffer are not being used.

2. The database system implements its buffer within the virtual memory provided by the operating system. Since the operating system knows about the memory requirements of all processes in the system, ideally it should be in charge of deciding what buffer blocks must be force-output to disk, and when. But, to ensure the write-ahead logging requirements in Section 17.7.1, the operating system should not write out the database buffer pages itself, but in-

rollback, instead of using compensation log records. This is because a crash may occur while a logical undo is in progress, and on recovery the system has to complete the logical undo; to do so, restart recovery will undo the partial effects of the earlier undo, using the physical undo information, and then perform the logical undo again, as we will see in Section 17.9.4.

At the end of the operation rollback, instead of generating a log record $< T_i, O_j,$ operation-end, $U >$, the system generates a log record $< T_i, O_j,$ operation-abort$>$.

2. When the backward scan of the log continues, the system skips all log records of the transaction until it finds the log record $<T_i, O_j,$ operation-begin$>$. After it finds the operation-begin log record, it processes log records of the transaction in the normal manner again.

Observe that skipping over physical log records when the operation-end log record is found during rollback ensures that the old values in the physical log record are not used for rollback, once the operation completes.

If the system finds a record $< T_i, O_j,$ operation-abort$>$, it skips all preceding records until it finds the record $< T_i, O_j,$ operation-begin$>$. These preceding log records must be skipped to prevent multiple rollback of the same operation, in case there had been a crash during an earlier rollback and the transaction had already been partly rolled back. When the transaction $T_i$ has been rolled back, the system adds a record $<T_i$ abort$>$ to the log.

If failures occur while a logical operation is in progress, the operation-end log record for the operation will not be found when the transaction is rolled back. However, for every update performed by the operation, undo information—in the form of the old value in the physical log records—is available in the log. The physical log records will be used to roll back the incomplete operation.



### 17.9.3 Checkpoints

Checkpointing is performed as described in Section 17.6. The system suspends updates to the database temporarily and carries out these actions:

1. It outputs to stable storage all log records currently residing in main memory.
2. It outputs to the disk all modified buffer blocks.
3. It outputs onto stable storage a log record <checkpoint $L$>, where $L$ is a list of all active transactions.

### 17.9.4 Restart Recovery

Recovery actions, when the database system is restarted after a failure, take place in two phases:

1. In the **redo phase**, the system replays updates of *all* transactions by scanning the log forward from the last checkpoint. The log records that are replayed include log records for transactions that were rolled back before sys-

- Log buffer, containing log records waiting to be output to the log on stable storage
- Cached query plans, which can be reused if the same query is submitted again

All database processes can access the data in shared memory. Since multiple processes may read or perform updates on data structures in shared memory, there must be a mechanism to ensure that only one of them is modifying any data structure at a time, and no process is reading a data structure while it is being written by others. Such **mutual exclusion** can be implemented by means of operating system functions called semaphores. Alternative implementations, with less overheads, use special **atomic instructions** supported by the computer hardware; one type of atomic instruction tests a memory location and sets it to 1 atomically. Further implementation details of mutual exclusion can be found in any standard operating system textbook. The mutual exclusion mechanisms are also used to implement latches.

To avoid the overhead of message passing, in many database systems, server processes implement locking by directly updating the lock table (which is in shared memory), instead of sending lock request messages to a lock manager process. The lock request procedure executes the actions that the lock manager process would take on getting a lock request. The actions on lock request and release are like those in Section 16.1.4, but with two significant differences:

- Since multiple server processes may access shared memory, mutual exclusion must be ensured on the lock table.
- If a lock cannot be obtained immediately because of a lock conflict, the lock request code keeps monitoring the lock table to check when the lock has been granted. The lock release code updates the lock table to note which process has been granted the lock.

  To avoid repeated checks on the lock table, operating system semaphores can be used by the lock request code to wait for a lock grant notification. The lock release code must then use the semaphore mechanism to notify waiting transactions that their locks have been granted.

Even if the system handles lock requests through shared memory, it still uses the lock manager process for deadlock detection.



### 18.2.2 Data Servers

Data-server systems are used in local-area networks, where there is a high-speed connection between the clients and the server, the client machines are comparable in processing power to the server machine, and the tasks to be executed are computation intensive. In such an environment, it makes sense to ship data to client machines, to perform all processing at the client machine (which may take a while), and then to ship the data back to the server machine. Note that this architecture requires full back-end functionality at the clients. Data-server architectures have been particularly popular in object-oriented database systems.

Interesting issues arise in such an architecture, since the time cost of communication between the client and the server is high compared to that of a local memory reference (milliseconds, versus less than 100 nanoseconds):

- **Page shipping versus item shipping**. The unit of communication for data can be of coarse granularity, such as a page, or fine granularity, such as a tuple (or an object, in the context of object-oriented database systems). We use the term **item** to refer to both tuples and objects.

  If the unit of communication is a single item, the overhead of message passing is high compared to the amount of data transmitted. Instead, when an item is requested, it makes sense also to send back other items that are likely to be used in the near future. Fetching items even before they are requested is called **prefetching**. Page shipping can be considered a form of prefetching if multiple items reside on a page, since all the items in the page are shipped when a process desires to access a single item in the page.

- **Locking**. Locks are usually granted by the server for the data items that it ships to the client machines. A disadvantage of page shipping is that client machines may be granted locks of too coarse a granularity—a lock on a page implicitly locks all items contained in the page. Even if the client is not accessing some items in the page, it has implicitly acquired locks on all prefetched items. Other client machines that require locks on those items may be blocked unnecessarily. Techniques for lock **de-escalation**, have been proposed where the server can request its clients to transfer back locks on prefetched items. If the client machine does not need a prefetched item, it can transfer locks on the item back to the server, and the locks can then be allocated to other clients.

- **Data caching**. Data that are shipped to a client on behalf of a transaction can be **cached** at the client, even after the transaction completes, if sufficient storage space is available. Successive transactions at the same client may be able to make use of the cached data. However, **cache coherency** is an issue: Even if a transaction finds cached data, it must make sure that those data are up to date, since they may have been updated by a different client after they were cached. Thus, a message must still be exchanged with the server to check validity of the data, and to acquire a lock on the data.

- **Lock caching**. If the use of data is mostly partitioned among the clients, with clients rarely requesting data that are also requested by other clients, locks can also be cached at the client machine. Suppose that a client finds a data item in the cache, and that it also finds the lock required for an access to the data item in the cache. Then, the access can proceed without any communication with the server. However, the server must keep track of cached locks; if a client requests a lock from the server, the server must **call back** all conflicting locks on the data item from any other client machines that have cached the locks. The task becomes more complicated when machine failures are taken into account. This technique differs from lock de-escalation in that lock caching takes place across transactions; otherwise, the two techniques are similar.

#### 18.3.3.1 Shared Memory

In a **shared-memory** architecture, the processors and disks have access to a common memory, typically via a bus or through an interconnection network. The benefit of shared memory is extremely efficient communication between processors—data in shared memory can be accessed by any processor without being moved with software. A processor can send messages to other processors much faster by using memory writes (which usually take less than a microsecond) than by sending a message through a communication mechanism. The downside of shared-memory machines is that the architecture is not scalable beyond 32 or 64 processors because the bus or the interconnection network becomes a bottleneck (since it is shared by all processors). Adding more processors does not help after a point, since the processors will spend most of their time waiting for their turn on the bus to access memory.

Shared-memory architectures usually have large memory caches at each processor, so that referencing of the shared memory is avoided whenever possible. However, at least some of the data will not be in the cache, and accesses will have to go to the shared memory. Moreover, the caches need to be kept coherent; that is, if a processor performs a write to a memory location, the data in that memory location should be either updated or removed from any processor where the data is cached. Maintaining cache coherency becomes an increasing overhead with increasing number of processors. Consequently, shared-memory machines are not capable of scaling up beyond a point; current shared-memory machines cannot support more than 64 processors.



#### 18.3.3.2 Shared Disk

In the **shared-disk** model, all processors can access all disks directly via an interconnection network, but the processors have private memories. There are two advantages of this architecture over a shared-memory architecture. First, since each processor has its own memory, the memory bus is not a bottleneck. Second, it offers a cheap way to provide a degree of **fault tolerance**: If a processor (or its memory) fails, the other processors can take over its tasks, since the database is resident on disks that are accessible from all processors. We can make the disk subsystem itself fault tolerant by using a RAID architecture, as described in Chapter 11. The shared-disk architecture has found acceptance in many applications.

The main problem with a shared-disk system is again scalability. Although the memory bus is no longer a bottleneck, the interconnection to the disk subsystem is now a bottleneck; it is particularly so in a situation where the database makes a large number of accesses to disks. Compared to shared-memory systems, shared-disk systems can scale to a somewhat larger number of processors, but communication across processors is slower (up to a few milliseconds in the absence of special-purpose hardware for communication), since it has to go through a communication network.

DEC clusters running Rdb were one of the early commercial users of the shared-disk database architecture. (Rdb is now owned by Oracle, and is called Oracle Rdb. Digital Equipment Corporation (DEC) is now owned by Compaq.)

to transfer $50 from account A-177 to account A-305, which is located at the Hillside branch, is a global transaction, since accounts in two different sites are accessed as a result of its execution.

In an ideal distributed database system, the sites would share a common global schema (although some relations may be stored only at some sites), all sites would run the same distributed database-management software, and the sites would be aware of each other's existence. If a distributed database is built from scratch, it would indeed be possible to achieve the above goals. However, in reality a distributed database has to be constructed by linking together multiple already-existing database systems, each with its own schema and possibly running different database-management software. Such systems are sometimes called **multidatabase systems** or **heterogeneous distributed database systems**. We discuss these systems in Section 19.8, where we show how to achieve a degree of global control despite the heterogeneity of the component systems.



### 18.4.2 Implementation Issues

Atomicity of transactions is an important issue in building a distributed database system. If a transaction runs across two sites, unless the system designers are careful, it may commit at one site and abort at another, leading to an inconsistent state. Transaction commit protocols ensure such a situation cannot arise. The *two-phase commit protocol* (2PC) is the most widely used of these protocols.

The basic idea behind 2PC is for each site to execute the transaction till just before commit, and then leave the commit decision to a single coordinator site; the transaction is said to be in the *ready* state at a site at this point. The coordinator decides to commit the transaction only if the transaction reaches the ready state at every site where it executed; otherwise (for example, if the transaction aborts at any site), the coordinator decides to abort the transaction. Every site where the transaction executed must follow the decision of the coordinator. If a site fails when a transaction is in ready state, when the site recovers from failure it should be in a position to either commit or abort the transaction, depending on the decision of the coordinator. The 2PC protocol is described in detail in Section 19.4.1.

Concurrency control is another issue in a distributed database. Since a transaction may access data items at several sites, transaction managers at several sites may need to coordinate to implement concurrency control. If locking is used (as is almost always the case in practice), locking can be performed locally at the sites containing accessed data items, but there is also a possibility of deadlock involving transactions originating at multiple sites. Therefore deadlock detection needs to be carried out across multiple sites. Failures are more common in distributed systems since not only may computers fail, but communication links may also fail. Replication of data items, which is the key to the continued functioning of distributed databases when failures occur, further complicates concurrency control. Section 19.5 provides detailed coverage of concurrency control in distributed databases.

The standard transaction models, based on multiple actions carried out by a single program unit, are often inappropriate for carrying out tasks that cross the boundaries of databases that cannot or will not cooperate to implement protocols such as 2PC.

in terms of their right to change schemas or database management system software. That software must also cooperate with other sites in exchanging information about transactions, to make transaction processing possible across multiple sites.

In contrast, in a **heterogeneous distributed database**, different sites may use different schemas, and different database management system software. The sites may not be aware of one another, and they may provide only limited facilities for cooperation in transaction processing. The differences in schemas are often a major problem for query processing, while the divergence in software becomes a hindrance for processing transactions that access multiple sites.

In this chapter, we concentrate on homogeneous distributed databases. However, in Section 19.8 we briefly discuss query processing issues in heterogeneous distributed database systems. Transaction processing issues in such systems are covered later, in Section 24.6.



## 19.2 Distributed Data Storage

Consider a relation $r$ that is to be stored in the database. There are two approaches to storing this relation in the distributed database:

- **Replication**. The system maintains several identical replicas (copies) of the relation, and stores each replica at a different site. The alternative to replication is to store only one copy of relation $r$.
- **Fragmentation**. The system partitions the relation into several fragments, and stores each fragment at a different site.

Fragmentation and replication can be combined: A relation can be partitioned into several fragments and there may be several replicas of each fragment. In the following subsections, we elaborate on each of these techniques.

### 19.2.1 Data Replication

If relation $r$ is replicated, a copy of relation $r$ is stored in two or more sites. In the most extreme case, we have **full replication**, in which a copy is stored in every site in the system.

There are a number of advantages and disadvantages to replication.

- **Availability**. If one of the sites containing relation $r$ fails, then the relation $r$ can be found in another site. Thus, the system can continue to process queries involving $r$, despite the failure of one site.
- **Increased parallelism**. In the case where the majority of accesses to the relation $r$ result in only the reading of the relation, then several sites can process queries involving $r$ in parallel. The more replicas of $r$ there are, the greater the chance that the needed data will be found in the site where the transaction is executing. Hence, data replication minimizes movement of data between sites.

#### 19.5.1.1 Single Lock-Manager Approach

In the **single lock-manager** approach, the system maintains a *single* lock manager that resides in a *single* chosen site—say $S_i$. All lock and unlock requests are made at site $S_i$. When a transaction needs to lock a data item, it sends a lock request to $S_i$. The lock manager determines whether the lock can be granted immediately. If the lock can be granted, the lock manager sends a message to that effect to the site at which the lock request was initiated. Otherwise, the request is delayed until it can be granted, at which time a message is sent to the site at which the lock request was initiated. The transaction can read the data item from *any* one of the sites at which a replica of the data item resides. In the case of a write, all the sites where a replica of the data item resides must be involved in the writing.

The scheme has these advantages:

- **Simple implementation**. This scheme requires two messages for handling lock requests, and one message for handling unlock requests.
- **Simple deadlock handling**. Since all lock and unlock requests are made at one site, the deadlock-handling algorithms discussed in Chapter 16 can be applied directly to this environment.

The disadvantages of the scheme are:

- **Bottleneck**. The site $S_i$ becomes a bottleneck, since all requests must be processed there.
- **Vulnerability**. If the site $S_i$ fails, the concurrency controller is lost. Either processing must stop, or a recovery scheme must be used so that a backup site can take over lock management from $S_i$, as described in Section 19.6.5.



#### 19.5.1.2 Distributed Lock Manager

A compromise between the advantages and disadvantages can be achieved through the **distributed lock-manager** approach, in which the lock-manager function is distributed over several sites.

Each site maintains a local lock manager whose function is to administer the lock and unlock requests for those data items that are stored in that site. When a transaction wishes to lock data item $Q$, which is not replicated and resides at site $S_i$, a message is sent to the lock manager at site $S_i$ requesting a lock (in a particular lock mode). If data item $Q$ is locked in an incompatible mode, then the request is delayed until it can be granted. Once it has determined that the lock request can be granted, the lock manager sends a message back to the initiator indicating that it has granted the lock request.

There are several alternative ways of dealing with replication of data items, which we study in Sections 19.5.1.3 to 19.5.1.6.

The distributed lock manager scheme has the advantage of simple implementation, and reduces the degree to which the coordinator is a bottleneck. It has a reasonably low overhead, requiring two message transfers for handling lock requests, and

| *A* | *B* | *C* |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 1 | 2 | 4 |
| 5 | 3 | 2 |
| 8 | 9 | 7 |

*r*

| *C* | *D* | *E* |
|---|---|---|
| 3 | 4 | 5 |
| 3 | 6 | 8 |
| 2 | 3 | 2 |
| 1 | 4 | 1 |
| 1 | 2 | 3 |

*s*

**Figure 19.7** Relations for Exercise 19.20.

**19.23** Describe how LDAP can be used to provide multiple hierarchical views of data, without replicating the base level data.

## Bibliographical Notes

Textbook discussions of distributed databases are offered by Ozsu and Valduriez [1999] and Ceri and Pelagatti [1984]. Computer networks are discussed in Tanenbaum [1996] and Halsall [1992]. Rothnie et al. [1977] was an early survey on distributed database systems. Breitbart et al. [1999b] presents an overview of distributed databases.

The implementation of the transaction concept in a distributed database are presented by Gray [1978], Traiger et al. [1982], Spector and Schwarz [1983], and Eppinger et al. [1991]. The 2PC protocol was developed by Lampson and Sturgis [1976] and Gray [1978]. The three-phase commit protocol is from Skeen [1981]. Mohan and Lindsay [1983] discuss two modified versions of 2PC, called *presume commit* and *presume abort*, that reduce the overhead of 2PC by defining default assumptions regarding the fate of transactions.

The bully algorithm in Section 19.6.5 is from Garcia-Molina [1982]. Distributed clock synchronization is discussed in Lamport [1978]. Distributed concurrency control is covered by Rosenkrantz et al. [1978], Bernstein et al. [1978], Bernstein et al. [1980b], Menasce et al. [1980], Bernstein and Goodman [1980], Bernstein and Goodman [1981a], Bernstein and Goodman [1982], and Garcia-Molina and Wiederhold [1982].

The transaction manager of R* is described in Mohan et al. [1986]. Concurrency control for replicated data that is based on the concept of voting is presented by Gifford [1979] and Thomas [1979]. Validation techniques for distributed concurrency-control schemes are described by Schlageter [1981], Ceri and Owicki [1983], and Bassiouni [1988]. Discussions of semantic-based transaction-management techniques are offered by Garcia-Molina [1983], Kumar and Stonebraker [1988] and Badrinath and Ramamritham [1992].

Attar et al. [1984] discusses the use of transactions in distributed recovery in database systems with replicated data. A survey of techniques for recovery in distributed database systems is presented by Kohler [1981].

Recently, the problem of concurrent updates to replicated data has re-emerged as an important research issue in the context of data warehouses. Problems in this

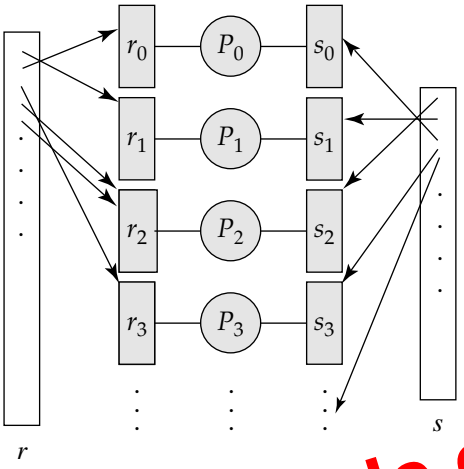



**Figure 20.2** Partitioned parallel join.



If one or both of the relations $r$ and $s$ are already partitioned on the join attributes (by either hash partitioning or range partitioning), the work needed for partitioning is reduced greatly. If the relations are not partitioned, or are partitioned on attributes other than the join attributes, then the tuples need to be repartitioned. Each processor $P_i$ reads in the tuples on disk $D_i$, computes for each tuple $t$ the partition $j$ to which $t$ belongs, and sends tuple $t$ to processor $P_j$. Processor $P_j$ stores the tuples on disk $D_j$.

We can optimize the join algorithm used locally at each processor to reduce I/O by buffering some of the tuples to memory, instead of writing them to disk. We describe such optimizations in Section 20.5.2.3.

Skew presents a special problem when range partitioning is used, since a partition vector that splits one relation of the join into equal-sized partitions may split the other relations into partitions of widely varying size. The partition vector should be such that $| r_i | + | s_i |$ (that is, the sum of the sizes of $r_i$ and $s_i$) is roughly equal over all the $i = 0, 1, \ldots, n - 1$. With a good hash function, hash partitioning is likely to have a smaller skew, except when there are many tuples with the same values for the join attributes.

### 20.5.2.2 Fragment-and-Replicate Join

Partitioning is not applicable to all types of joins. For instance, if the join condition is an inequality, such as $r \bowtie_{r.a<s.b} s$, it is possible that all tuples in $r$ join with some tuple in $s$ (and vice versa). Thus, there may be no easy way of partitioning $r$ and $s$ so that tuples in partition $r_i$ join with only tuples in partition $s_i$.

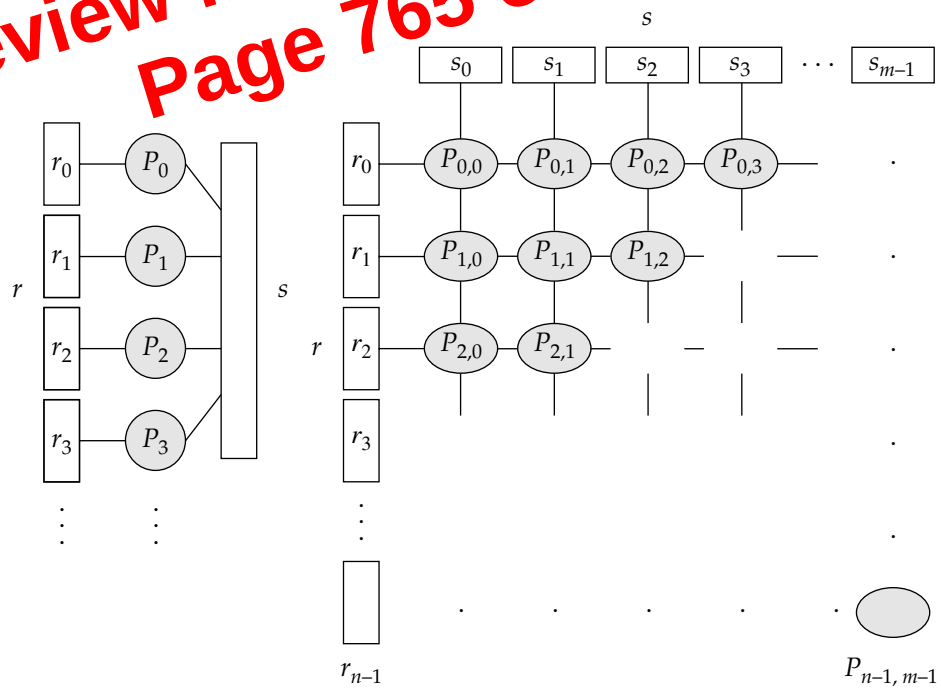
We can parallelize such joins by using a technique called *fragment and replicate*. We first consider a special case of fragment and replicate—**asymmetric fragment-and-replicate join**—which works as follows.

1. The system partitions one of the relations—say, $r$. Any partitioning technique can be used on $r$, including round-robin partitioning.
2. The system replicates the other relation, $s$, across all the processors.
3. Processor $P_i$ then locally computes the join of $r_i$ with all of $s$, using any join technique.

The asymmetric fragment-and-replicate scheme appears in Figure 20.3a. If $r$ is already stored by partitioning, there is no need to partition it further in step 1. All that is required is to replicate $s$ across all processors.

The general case of **fragment and replicate join** appears in Figure 20.3b; it works this way: The system partitions relation $r$ into $n$ partitions, $r_0, r_1, \ldots, r_{n-1}$, and partitions $s$ into $m$ partitions, $s_0, s_1, \ldots, s_{m-1}$. As before, any partitioning technique may be used on $r$ and on $s$. The values of $m$ and $n$ do not need to be equal, but they must be chosen so that there are at least $m * n$ processors. Asymmetric fragment and replicate is simply a special case of general fragment and replicate, where $m = 1$. Fragment and replicate reduces the sizes of the relations at each processor, compared to asymmetric fragment and replicate.






(a) Asymmetric fragment and replicate

(b) Fragment and replicate

**Figure 20.3** Fragment-and-replicate schemes.

A large parallel database system must also address these availability issues:

- Resilience to failure of some processors or disks
- Online reorganization of data and schema changes

We consider these issues here.

With a large number of processors and disks, the probability that at least one processor or disk will malfunction is significantly greater than in a single-processor system with one disk. A poorly designed parallel system will stop functioning if any component (processor or disk) fails. Assuming that the probability of failure of a single processor or disk is small, the probability of failure of the system goes up linearly with the number of processors and disks. If a single processor or disk would fail once every 5 years, a system with 100 processors would have a failure every 18 days.

Therefore, large-scale parallel database systems, such as Compaq Himalaya, Teradata, and Informix XPS (now a division of IBM), are designed to operate even if a processor or disk fails. Data are replicated across at least two processors. If a processor fails, the data that it stored can still be accessed from the other processors. The system keeps track of failed processors and distributes the work among functioning processors. Requests for data stored at the failed site are automatically routed to the backup sites that store a replica of the data. If all the data of a processor $A$ are replicated at a single processor $B$, $B$ will have to handle all the requests to $A$ as well as those to itself, and that will result in $B$ becoming a bottleneck. Therefore, the replicas of the data of a processor are partitioned across multiple other processors.

When we are dealing with large volumes of data (ranging in the terabytes), simple operations, such as creating indices, and changes to schema, such as adding a column to a relation, can take a long time — perhaps hours or even days. Therefore, it is unacceptable for the database system to be unavailable while such operations are in progress. Many parallel database systems, such as the Compaq Himalaya systems, allow such operations to be performed **online**, that is, while the system is executing other transactions.

Consider, for instance, **online index construction**. A system that supports this feature allows insertions, deletions, and updates on a relation even as an index is being built on the relation. The index-building operation therefore cannot lock the entire relation in shared mode, as it would have done otherwise. Instead, the process keeps track of updates that occur while it is active, and incorporates the changes into the index being constructed.

## 20.8 Summary

- Parallel databases have gained significant commercial acceptance in the past 15 years.
- In I/O parallelism, relations are partitioned among available disks so that they can be retrieved faster. Three commonly used partitioning techniques are round-robin partitioning, hash partitioning, and range partitioning.

- Range query
- Skew
  - Execution skew
  - Attribute-value skew
  - Partition skew
- Handling of skew
  - Balanced range-partitioning vector
  - Histogram
  - Virtual processors
- Interquery parallelism
- Cache coherency
- Intraquery parallelism
  - Intraoperation parallelism
  - Interoperation parallelism
- Parallel sort
  - Range-partitioning sort
  - Parallel external sort–merge
- Data parallelism
- Parallel join
  - Partitioned join
  - Fragment-and-replicate join
  - Asymmetric fragment-and-replicate join
  - Partitioned parallel hash–join
  - Parallel nested-loop join
- Parallel selection
- Parallel duplicate elimination
- Parallel projection
- Parallel aggregation
- Cost of parallel evaluation
- Interoperation parallelism
  - Pipelined parallelism
  - Independent parallelism
- Query optimization
- Scheduling
- Exchange-operator model
- Design of parallel systems
- Online index construction



## Exercises

**20.1** For each of the three partitioning techniques, namely round-robin, hash partitioning, and range partitioning, give an example of a query for which that partitioning technique would provide the fastest response.

**20.2** In a range selection on a range-partitioned attribute, it is possible that only one disk may need to be accessed. Describe the benefits and drawbacks of this property.

**20.3** What factors could result in skew when a relation is partitioned on one of its attributes by:

**a.** Hash partitioning
**b.** Range partitioning

In each case, what can be done to reduce the skew?

**20.4** What form of parallelism (interquery, interoperation, or intraoperation) is likely to be the most important for each of the following tasks.

**a.** Increasing the throughput of a system with many small queries
**b.** Increasing the throughput of a system with a few large queries, when the number of disks and processors is large

### 21.2.2 Tunable Parameters

Database administrators can tune a database system at three levels. The lowest level is at the hardware level. Options for tuning systems at this level include adding disks or using a RAID system if disk I/O is a bottleneck, adding more memory if the disk buffer size is a bottleneck, or moving to a faster processor if CPU use is a bottleneck.

The second level consists of the database-system parameters, such as buffer size and checkpointing intervals. The exact set of database-system parameters that can be tuned depends on the specific database system. Most database-system manuals provide information on what database-system parameters can be adjusted, and how you should choose values for the parameters. Well-designed database systems perform as much tuning as possible automatically, freeing the user or database administrator from the burden. For instance, in many database systems the buffer size is fixed but tunable. If the system automatically adjusts the buffer size by observing indicators such as page-fault rates, then the user will not have to worry about tuning the buffer size.

The third level is the highest level. It includes the schema and transactions. The administrator can tune the design of the schema, the indices that are created, and the transactions that are executed, to improve performance. Tuning at this level is comparatively system independent.

The three levels of tuning interact with one another; we must consider them together when tuning a system. For example, tuning at a higher level may result in the hardware bottleneck changing from the disk system to the CPU, or vice versa.



### 21.2.3 Tuning of Hardware

Even in a well-designed transaction processing system, each transaction usually has to do at least a few I/O operations, if the data required by the transaction is on disk. An important factor in tuning a transaction processing system is to make sure that the disk subsystem can handle the rate at which I/O operations are required. For instance, disks today have an access time of about 10 milliseconds, and transfer times of 20 MB per second, which gives about 100 random access I/O operations of 1 KB each. If each transaction requires just 2 I/O operations, a single disk would support at most 50 transactions per second. The only way to support more transactions per second is to increase the number of disks. If the system needs to support $n$ transactions per second, each performing 2 I/O operations, data must be striped (or otherwise partitioned) across $n/50$ disks (ignoring skew).

Notice here that the limiting factor is not the capacity of the disk, but the speed at which random data can be accessed (limited in turn by the speed at which the disk arm can move). The number of I/O operations per transaction can be reduced by storing more data in memory. If all data are in memory, there will be no disk I/O except for writes. Keeping frequently used data in memory reduces the number of disk I/Os, and is worth the extra cost of memory. Keeping very infrequently used data in memory would be a waste, since memory is much more expensive than disk.

The question is, for a given amount of money available for spending on disks or memory, what is the best way to spend the money to achieve maximum number of

There are many data sources that are not relational databases, and in fact may not be databases at all. Examples are flat files and email stores. Microsoft's **OLE-DB** is a C++ API with goals similar to ODBC, but for nondatabase data sources that may provide only limited querying and update facilities. Just like ODBC, OLE-DB provides constructs for connecting to a data source, starting a session, executing commands, and getting back results in the form of a rowset, which is a set of result rows.

However, OLE-DB differes from ODBC in several ways. To support data sources with limited feature support, features in OLE-DB are divided into a number of interfaces, and a data source may implement only a subset of the interfaces. An OLE-DB program can negotiate with a data source to find what interfaces are supported. In ODBC commands are always in SQL. In OLE-DB, commands may be in any language supported by the data source; while some sources may support SQL, or a limited subset of SQL, other sources may provide only simple capabilities such as accessing data in a flat file, without any query capability. Another major difference of OLE-DB from ODBC is that a rowset is an object that can be shared by multiple applications through shared memory. A rowset object can be updated by one application, and other applications sharing that object would get notified about the change.

The **Active Data Objects (ADO) API**, also created by Microsoft, provides an easy-to-use interface to the OLE-DB functionality, which can be called from scripting languages, such as VBScript and JScript.



### 21.4.3 Object-Database Standards

Standards in the area of object-oriented databases have so far been driven primarily by OODB vendors. The *Object Database Management Group* (ODMG) is a group formed by OODB vendors to standardize the data model and language interfaces to OODBs. The C++ language interface specified by ODMG was discussed in Chapter 8. The ODMG has also specified a Java interface and a Smalltalk interface.

The *Object Management Group* (OMG) is a consortium of companies, formed with the objective of developing a standard architecture for distributed software applications based on the object-oriented model. OMG brought out the *Object Management Architecture* (OMA) reference model. The *Object Request Broker* (ORB) is a component of the OMA architecture that provides message dispatch to distributed objects transparently, so the physical location of the object is not important. The **Common Object Request Broker Architecture** (**CORBA**) provides a detailed specification of the ORB, and includes an **Interface Description Language** (**IDL**), which is used to define the data types used for data interchange. The IDL helps to support data conversion when data are shipped between systems with different data representations.

### 21.4.4 XML-Based Standards

A wide variety of standards based on XML (see Chapter 10) have been defined for a wide variety of applications. Many of these standards are related to e-commerce. They include standards promulgated by nonprofit consortia and corporate-backed efforts to create defacto standards. RosettaNet, which falls into the former category, uses XML-based standards to facilitate supply-chain management in the computer

**21.6** Suppose a system runs three types of transactions. Transactions of type A run at the rate of 50 per second, transactions of type B run at 100 per second, and transactions of type C run at 200 per second. Suppose the mix of transactions has 25 percent of type A, 25 percent of type B, and 50 percent of type C.

**a.** What is the average transaction throughput of the system, assuming there is no interference between the transactions.

**b.** What factors may result in interference between the transactions of different types, leading to the calculated throughput being incorrect?

**21.7** Suppose the price of memory falls by half, and the speed of disk access (number of accesses per second) doubles, while all other factors remain the same. What would be the effect of this change on the 5 minute and 1 minute rule?

**21.8** List some of the features of the TPC benchmarks that help make them realistic and dependable measures.

**21.9** Why was the TPC-D benchmark replaced by the TPC-H and TPC-R benchmarks?

**21.10** List some benefits and drawbacks of an anticipatory standard compared to a reactionary standard.

**21.11** Suppose someone impersonates a company and gets a certificate from a certificate-issuing authority. What is the effect on things (such as purchase orders or programs) certified by the impersonated company, and on things certified by other companies?



## Project Suggestions

Each of the following is a large project, which can be a semester-long project done by a group of students. The difficulty of the project can be adjusted easily by adding or deleting features.

**Project 21.1** Consider the E-R schema of Exercise 2.7 (Chapter 2), which represents information about teams in a league. Design and implement a Web-based system to enter, update, and view the data.

**Project 21.2** Design and implement a shopping cart system that lets shoppers collect items into a shopping cart (you can decide what information is to be supplied for each item) and purchased together. You can extend and use the E-R schema of Exercise 2.12 of Chapter 2. You should check for availability of the item and deal with nonavailable items as you feel appropriate.

**Project 21.3** Design and implement a Web-based system to record student registration and grade information for courses at a university.

**Project 21.4** Design and implement a system that permits recording of course performance information—specifically, the marks given to each student in each assignment or exam of a course, and computation of a (weighted) sum of marks to get the total course marks. The number of assignments/exams should not

CHAPTER 22

# Advanced Querying and Information Retrieval



Businesses have begun to exploit the burgeoning data online to make better decisions about their activities, such as what items to stock and how best to target customers to increase sales. Many of their queries are rather complicated, however, and certain types of information cannot be extracted even by using SQL.

Several techniques and tools are available to help with decision support. Several tools for data analysis allow analysts to view data in different ways. Other analysis tools precompute summaries of very large amounts of data, in order to give fast responses to queries. The SQL:1999 standard now contains additional constructs to support data analysis. Another approach to getting knowledge from data is to use *data mining*, which aims at detecting various types of patterns in large volumes of data. Data mining supplements various types of statistical techniques with similar goals.

Textual data, too, has grown explosively. Textual data is unstructured, unlike the rigidly structured data in relational databases. Querying of unstructured textual data is referred to as *information retrieval*. Information retrieval systems have much in common with database systems—in particular, the storage and retrieval of data on secondary storage. However, the emphasis in the field of information systems is different from that in database systems, concentrating on issues such as querying based on keywords; the relevance of documents to the query; and the analysis, classification, and indexing of documents.

This chapter covers decision support, including online analytical processing and data mining and information retrieval.

## 22.1 Decision-Support Systems

Database applications can be broadly classified into transaction processing and decision support, as we have seen earlier in Section 21.3.2. Transaction-processing systems are widely used today, and companies have accumulated a vast amount of information generated by these systems.

### 22.3.2 Classification

As mentioned in Section 22.3.1, prediction is one of the most important types of data mining. We outline what is classification, study techniques for building one type of classifiers, called decision tree classifiers, and then study other prediction techniques.

Abstractly, the **classification** problem is this: Given that items belong to one of several classes, and given past instances (called **training instances**) of items along with the classes to which they belong, the problem is to predict the class to which a new item belongs. The class of the new instance is not known, so other attributes of the instance must be used to predict the class.

Classification can be done by finding rules that partition the given data into disjoint groups. For instance, suppose that a credit-card company wants to decide whether or not to give a credit card to an applicant. The company has a variety of information about the person, such as her age, educational background, annual income, and current debts, that it can use for making a decision.

Some of this information could be relevant to the credit worthiness of the applicant, whereas some may not be. To make this decision, the company assigns a credit-worthiness level of excellent, good, average, or bad to each of a sample set of *current* customers according to each customer's payment history. Then, the company attempts to find rules that classify its current customers into excellent, good, average, or bad, on the basis of the information about the person, other than the actual payment history (which is unavailable for new customers). Let us consider just two attributes: education level (highest degree earned) and income. The rules may be of the following form:

$$\forall person\ \ P,\ P.degree = masters \textbf{ and } P.income > 75,000 \Rightarrow P.credit = excellent$$

$$\forall\ person\ \ P,\ P.degree = bachelors \textbf{ or } (P.income \geq 25,000 \textbf{ and } P.income \leq 75,000) \Rightarrow P.credit = good$$

Similar rules would also be present for the other credit worthiness levels (average and bad).

The process of building a classifier starts from a sample of data, called a **training set**. For each tuple in the training set, the class to which the tuple belongs is already known. For instance, the training set for a credit-card application may be the existing customers, with their credit worthiness determined from their payment history. The actual data, or population, may consist of all people, including those who are not existing customers. There are several ways of building a classifier, as we shall see.

#### 22.3.2.1 Decision Tree Classifiers

The decision tree classifier is a widely used technique for classification. As the name suggests, **decision tree classifiers** use a tree; each leaf node has an associated class, and each internal node has a predicate (or more generally, a function) associated with it. Figure 22.6 shows an example of a decision tree.

To classify a new instance, we start at the root, and traverse the tree to reach a leaf; at an internal node we evaluate the predicate (or function) on the data instance,

all items in the set are contained in the purchase. For instance, if a purchase included items $a$, $b$, and $c$, counts would be incremented for $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{b, c\}$, $\{a, c\}$, and $\{a, b, c\}$. Those sets with a sufficiently high count at the end of the pass correspond to items that have a high degree of association.

The number of sets grows exponentially, making the procedure just described infeasible if the number of items is large. Luckily, almost all the sets would normally have very low support; optimizations have been developed to eliminate most such sets from consideration. These techniques use multiple passes on the database, considering only some sets in each pass.

In the **a priori** technique for generating large itemsets, only sets with single items are considered in the first pass. In the second pass, sets with two items are considered, and so on.

At the end of a pass all sets with sufficient support are output as large itemsets. Sets found to have too little support at the end of a pass are eliminated. Once a set is eliminated, none of its supersets needs to be considered. In other words, in pass $i$ we need to count only supports for sets of size $i$ such that all subsets of the set have been found to have sufficiently high support; it suffices to test all subsets of size $i - 1$ to ensure this property. At the end of some pass $i$, we would find that no set of size $i$ has sufficient support, so we do not need to consider any set of size $i + 1$. Computation then terminates.



### 22.3.4 Other Types of Associations

Using plain association rules has several shortcomings. One of the major shortcomings is that many associations are not very interesting, since they can be predicted. For instance, if many people buy cereal and many people buy bread, we can predict that a fairly large number of people would buy both, even if there is no connection between the two purchases. What would be interesting is a **deviation** from the expected co-occurrence of the two. In statistical terms, we look for **correlations** between items; correlations can be positive, in that the co-occurrence is higher than would have been expected, or negative, in that the items co-occur less frequently than predicted. See a standard textbook on statistics for more information about correlations.

Another important class of data-mining applications is sequence associations (or correlations). Time-series data, such as stock prices on a sequence of days, form an example of sequence data. Stock-market analysts want to find associations among stock-market price sequences. An example of such a association is the following rule: "Whenever bond rates go up, the stock prices go down within 2 days." Discovering such association between sequences can help us to make intelligent investment decisions. See the bibliographical notes for references to research on this topic.

Deviations from temporal patterns are often interesting. For instance, if a company has been growing at a steady rate each year, a deviation from the usual growth rate is surprising. If sales of winter clothes go down in summer, it is not surprising, since we can predict it from past years; a deviation that we could not have predicted from past experience would be considered interesting. Mining techniques can find deviations from what one would have expected on the basis of past temporal/sequential patterns. See the bibliographical notes for references to research on this topic.

Unless updates at the sources are replicated at the warehouse via two-phase commit, the warehouse will never be quite up to date with the sources. Two-phase commit is usually far too expensive to be an option, so data warehouses typically have slightly out-of-date data. That, however, is usually not a problem for decision-support systems.

- **What schema to use.** Data sources that have been constructed independently are likely to have different schemas. In fact, they may even use different data models. Part of the task of a warehouse is to perform schema integration, and to convert data to the integrated schema before they are stored. As a result, the data stored in the warehouse are not just a copy of the data at the sources. Instead, they can be thought of as a materialized view of the data at the sources.
- **Data cleansing.** The task of correcting and preprocessing data is called **data cleansing**. Data sources often deliver data with numerous minor inconsistencies, that can be corrected. For example, names are often misspelled, and addresses may have street/area/city names misspelled, or zip codes entered incorrectly. These can be corrected to a reasonable extent by consulting a database of street names and zip codes in each city. Address lists collected from multiple sources may have duplicates that need to be eliminated in a **merge–purge operation**. Records for multiple individuals in a house may be grouped together so only one mailing is sent to each house; this operation is called **householding**.
- **How to propagate updates**. Updates on relations at the data sources must be propagated to the data warehouse. If the relations at the data warehouse are exactly the same as those at the data source, the propagation is straightforward. If they are not, the problem of propagating updates is basically the *view-maintenance* problem, which was discussed in Section 14.5.
- **What data to summarize.** The raw data generated by a transaction-processing system may be too large to store online. However, we can answer many queries by maintaining just summary data obtained by aggregation on a relation, rather than maintaining the entire relation. For example, instead of storing data about every sale of clothing, we can store total sales of clothing by item-name and category.

  Suppose that a relation $r$ has been replaced by a summary relation $s$. Users may still be permitted to pose queries as though the relation $r$ were available online. If the query requires only summary data, it may be possible to transform it into an equivalent one using $s$ instead; see Section 14.5.



## 22.4.2 Warehouse Schemas

Data warehouses typically have schemas that are designed for data analysis, using tools such as OLAP tools. Thus, the data are usually multidimensional data, with dimension attributes and measure attributes. Tables containing multidimensional data are called **fact tables** and are usually very large. A table recording sales information

Given a query $Q$, the job of an information retrieval system is to return documents in descending order of their relevance to $Q$. Since there may be a very large number of documents that are relevant, information retrieval systems typically return only the first few documents with the highest degree of estimated relevance, and permit users to interactively request further documents.

### 22.5.1.2 Relevance Using Hyperlinks

Early Web search engines ranked documents by using only relevance measures similar to those described in Section 22.5.1.1. However, researchers soon realized that Web documents have information that plain text documents do not have, namely hyperlinks. And in fact, the relevance ranking of a document is affected more by hyperlinks that point *to* the document, than by hyperlinks going out of the document.

The basic idea of site ranking is to find sites that are popular, and to rank pages from such sites higher than pages from other sites. A site is identified by the internet address part of the URL, such as www.bell-labs.com in a URL http://www.bell-labs.com/topic/books/db-book. A site usually contains multiple Web pages. Since most searches are intended to find information from popular sites, ranking pages from popular sites higher is generally a good idea. For instance, the term "google" may occur in vast numbers of pages, but the site google.com is the most popular among the sites with pages that contain the term "google". Documents from google.com containing the term "google" would therefore be ranked as the most relevant to the term "google".



This raises the question of how to define the popularity of a site. One way would be to find how many times a site is accessed. However, getting such information is impossible without the cooperation of the site, and is infeasible for a Web search engine to implement. A very effective alternative uses hyperlinks; it defines $p(s)$, the **popularity of a site** $s$, as the number of sites that contain at least one page with a link to site $s$.

Traditional measures of relevance of the page (which we saw in Section 22.5.1.2) can be combined with the popularity of the site containing the page to get an overall measure of the relevance of the page. Pages with high overall relevance value are returned as answers to a query, as before.

Note also that we used the popularity of a *site* as a measure of relevance of individual pages at the site, not the popularity of individual *pages*. There are at least two reasons for this. First, most sites contain only links to root pages of other sites, so all other pages would appear to have almost zero popularity, when in fact they may be accessed quite frequently by following links from the root page. Second, there are far fewer sites than pages, so computing and using popularity of sites is cheaper than computing and using popularity of pages.

There are more refined notions of popularity of sites. For instance, a link from a popular site to another site $s$ may be considered to be a better indication of the popularity of $s$ than a link to $s$ from a less popular site.[6] This notion of popularity

6. This is similar in some sense to giving extra weight to endorsements of products by celebrities (such as film stars), so its significance is open to question!

is in fact circular, since the popularity of a site is defined by the popularity of other sites, and there may be cycles of links between sites. However, the popularity of sites can be defined by a system of simultaneous linear equations, which can be solved by matrix manipulation techniques. The linear equations are defined in such a way that they have a unique and well-defined solution.

The popular Web search engine google.com uses the referring-site popularity idea in its definition **page rank**, which is a measure of popularity of a page. This approach of ranking of pages gave results so much better than previously used ranking techniques, that google.com became a widely used search engine, in a rather short period of time.

There is another, somewhat similar, approach, derived interestingly from a theory of social networking developed by sociologists in the 1950s. In the social networking context, the goal was to define the prestige of people. For example, the president of the United States has high prestige since a large number of people know him. If someone is known by multiple prestigious people, then she also has high prestige, even if she is not known by as large a number of people.

The above idea was developed into a notion of *hubs* and *authorities* that takes into account the presence of directories that link to pages containing useful information. A **hub** is a page that stores links to many pages; it does not in itself contain actual information on a topic, but points to pages that contain actual information. In contrast, an **authority** is a page that contains actual information on a topic, although it may not be directly pointed to by many pages. Each page then gets a prestige value as a hub (*hub-prestige*), and another prestige value as an authority (*authority-prestige*). The definitions of prestige, as before, are cyclic and are defined by a set of simultaneous linear equations. A page gets higher hub-prestige if it points to many pages with high authority-prestige, while a page gets higher authority-prestige if it is pointed to by many pages with high hub-prestige. Given a query, pages with highest authority-prestige are ranked higher than other pages. See the bibliographical notes for references giving further details.



### 22.5.1.3 Similarity-Based Retrieval

Certain information-retrieval systems permit **similarity-based retrieval**. Here, the user can give the system document $A$, and ask the system to retrieve documents that are "similar" to $A$. The similarity of a document to another may be defined, for example, on the basis of common terms. One approach is to find $k$ terms in $A$ with highest values of $r(d, t)$, and to use these $k$ terms as a query to find relevance of other documents. The terms in the query are themselves weighted by $r(d, t)$.

If the set of documents similar to $A$ is large, the system may present the user a few of the similar documents, allow him to choose the most relevant few, and start a new search based on similarity to $A$ *and* to the chosen documents. The resultant set of documents is likely to be what the user intended to find.

The same idea is also used to help users who find many documents that appear to be relevant on the basis of the keywords, but are not. In such a situation, instead of adding further keywords to the query, users may be allowed to identify one or a few of the returned documents as relevant; the system then uses the identified documents

- False positives may occur because irrelevant documents get higher rankings than relevant documents. This too depends on how many documents are examined. One option is to measure precision as a function of number of documents fetched.

A better and more intuitive alternative for measuring precision is to measure it as a function of recall. With this combined measure, both precision and recall can be computed as a function of number of documents, if required.

For instance, we can say that with a recall of 50 percent the precision was 75 percent, whereas at a recall of 75 percent the precision dropped to 60 percent. In general, we can draw a graph relating precision to recall. These measures can be computed for individual queries, then averaged out across a suite of queries in a query benchmark.

Yet another problem with measuring precision and recall lies in how to define which documents are really relevant and which are not. In fact, it requires understanding of natural language, and understanding of the intent of the query, to decide if a document is relevant or not. Researchers therefore have created collections of documents and queries, and have manually tagged documents as relevant or irrelevant to the queries. Different ranking systems can be run on these collections to measure their average precision and recall across multiple queries.



### 22.5.4 Web Search Engines

**Web crawlers** are programs that locate and gather information on the Web. They recursively follow hyperlinks present in known documents to find other documents. A crawler retrieves the documents and adds information found in the documents to a combined index; the document is generally not stored, although some search engines do cache a copy of the document to give clients faster access to the documents.

Since the number of documents on the Web is very large, it is not possible to crawl the whole Web in a short period of time; and in fact, all search engines cover only some portions of the Web, not all of it, and their crawlers may take weeks or months to perform a single crawl of all the pages they cover. There are usually many processes, running on multiple machines, involved in crawling. A database stores a set of links (or sites) to be crawled; it assigns links from this set to each crawler process. New links found during a crawl are added to the database, and may be crawled later if they are not crawled immediately. Pages found during a crawl are also handed over to an indexing system, which may be running on a different machine. Pages have to be refetched (that is, links recrawled) periodically to obtain updated information, and to discard sites that no longer exist, so that the information in the search index is kept reasonably up to date.

The indexing system itself runs on multiple machines in parallel. It is not a good idea to add pages to the same index that is being used for queries, since doing so would require concurrency control on the index, and affect query and update performance. Instead, one copy of the index is used to answer queries while another copy is updated with newly crawled pages. At periodic intervals the copies switch over, with the old one being updated while the new copy is being used for queries.

one that libraries use, and, when it displays a particular document, it can also display a brief description of documents that are close in the hierarchy.

In an information retrieval system, there is no need to keep a document in a single spot in the hierarchy. A document that talks of mathematics for computer scientists could be classified under mathematics as well as under computer science. All that is stored at each spot is an identifier of the document (that is, a pointer to the document), and it is easy to fetch the contents of the document by using the identifier.

As a result of this flexibility, not only can a document be classified under two locations, but also a subarea in the classification hierarchy can itself occur under two areas. The class of "graph algorithm" document can appear both under mathematics and under computer science. Thus, the classification hierarchy is now a directed acyclic graph (DAG), as shown in Figure 22.11. A graph-algorithm document may appear in a single location in the DAG, but can be reached via multiple paths.

A **directory** is simply a classification DAG structure. Each leaf of the directory stores links to documents on the topic represented by the leaf. Internal nodes may also contain links, for example to documents that cannot be classified under any of the child nodes.

To find information on a topic, a user would start at the root of the directory and follow paths down the DAG until reaching a node representing the desired topic. While browsing down the directory, the user can find not only documents on the topic he is interested in, but also find related documents and related classes in the classification hierarchy. The user may learn new information by browsing through documents (or subclasses) within the related classes.

Organizing the enormous amount of information available on the Web into a directory structure is a daunting task.



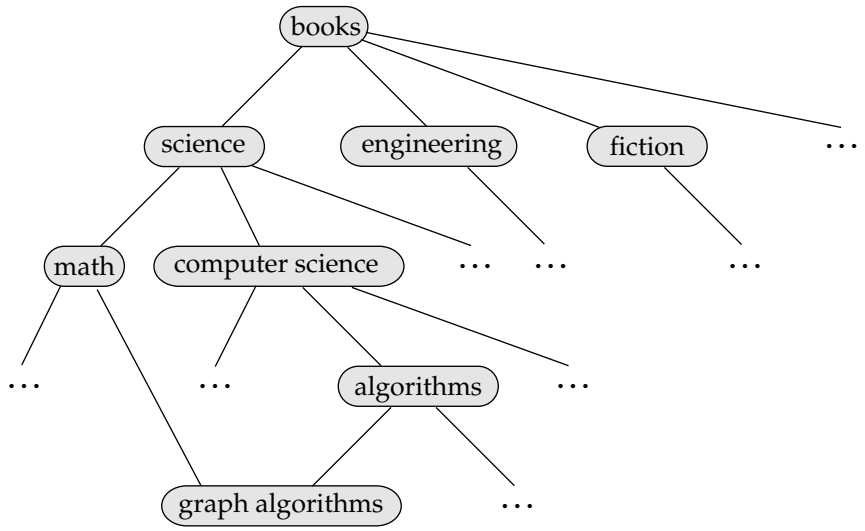



**Figure 22.11** A classification DAG for a library information retrieval system.

this period starts. This notion differs from the notion of interval we used previously, which refers to an interval of time with specific starting and ending times.[1]

### 23.2.2 Temporal Query Languages

A database relation without temporal information is sometimes called a **snapshot relation**, since it reflects the state in a snapshot of the real world. Thus, a snapshot of a temporal relation at a point in time $t$ is the set of tuples in the relation that are true at time $t$, with the time-interval attributes projected out. The snapshot operation on a temporal relation gives the snapshot of the relation at a specified time (or the current time, if the time is not specified).

A **temporal selection** is a selection that involves the time attributes; a **temporal projection** is a projection where the tuples in the projection inherit their times from the tuples in the original relation. A **temporal join** is a join, with the time of a tuple in the result being the intersection of the times of the tuples from which it is derived. If the times do not intersect, the tuple is removed from the result.

The predicates *precedes*, *overlaps*, and *contains* can be applied on intervals; their meanings should be clear. The *intersect* operation can be applied on two intervals, to give a single (possibly empty) interval. However, the union of two intervals may or may not be a single interval.

Functional dependencies must be used with care in a temporal relation. Although the account number may functionally determine the balance at any given point in time, obviously the balance can change over time. A **temporal functional dependency** $X \xrightarrow{\tau} Y$ holds on a relation schema $R$ if, for all legal instances $r$ of $R$, all snapshots of $r$ satisfy the functional dependency $X \rightarrow Y$.

Several proposals have been made for extending SQL to improve its support of temporal data. SQL:1999 Part 7 (SQL/Temporal), which is currently under development, is the proposed standard for temporal extensions to SQL.

## 23.3 Spatial and Geographic Data

Spatial data support in databases is important for efficiently storing, indexing, and querying of data based on spatial locations. For example, suppose that we want to store a set of polygons in a database, and to query the database to find all polygons that intersect a given polygon. We cannot use standard index structures, such as B-trees or hash indices, to answer such a query efficiently. Efficient processing of the above query would require special-purpose index structures, such as R-trees (which we study later) for the task.

Two types of spatial data are particularly important:

- **Computer-aided-design (CAD) data**, which includes spatial information about how objects—such as buildings, cars, or aircraft—are constructed. Other important examples of computer-aided-design databases are integrated-circuit and electronic-device layouts.

1. Many temporal database researchers feel this type should have been called **span** since it does not specify an exact start or end time, only the time span between the two.

- **Geographic data** such as road maps, land-usage maps, topographic elevation maps, political maps showing boundaries, land ownership maps, and so on. **Geographic information systems** are special-purpose databases tailored for storing geographic data.

Support for geographic data has been added to many database systems, such as the IBM DB2 Spatial Extender, the Informix Spatial Datablade, and Oracle Spatial.

### 23.3.1 Representation of Geometric Information

Figure 23.2 illustrates how various geometric constructs can be represented in a database, in a normalized fashion. We stress here that geometric information can be represented in several different ways, only some of which we describe.

A *line segment* can be represented by the coordinates of its endpoints. For example, in a map database, the two coordinates of a point would be its latitude and longi-



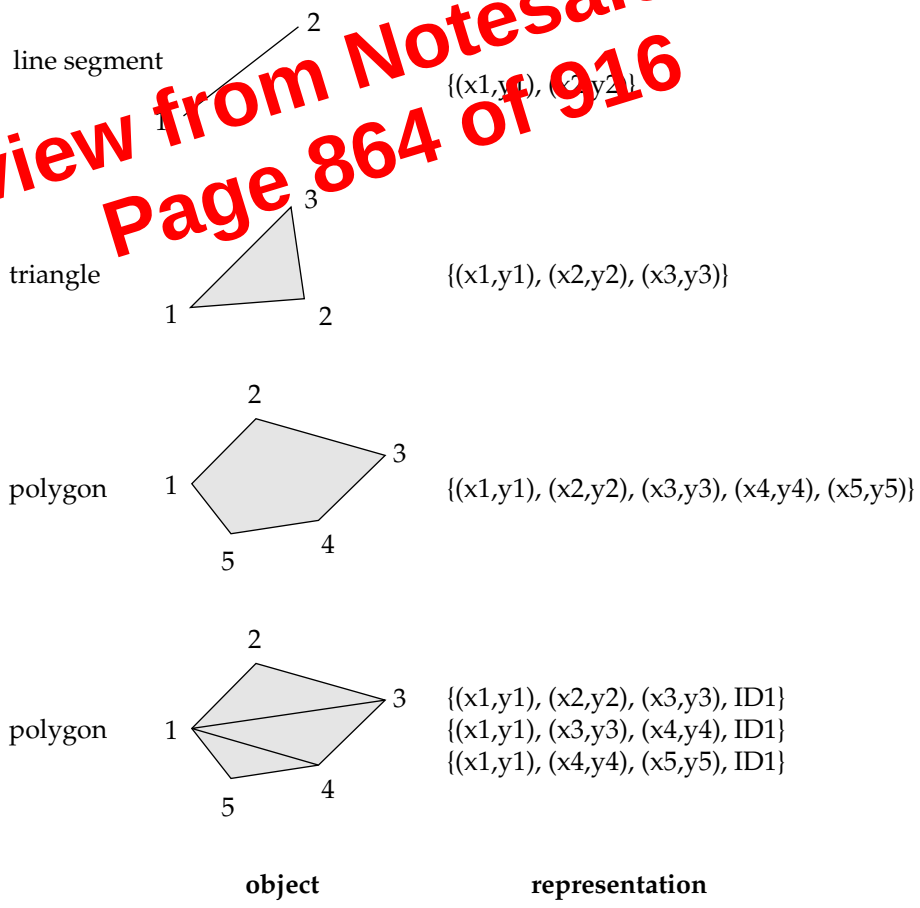



**Figure 23.2** Representation of geometric constructs.

in two. Points that lie in the left partition go into the left subtree; points that lie in the right partition go into the right subtree. In a balanced binary tree, the partition is chosen so that approximately one-half of the points stored in the subtree fall in each partition. Similarly, each level of a B-tree splits a one-dimensional interval into multiple parts.

We can use that intuition to create tree structures for two-dimensional space, as well as in higher-dimensional spaces. A tree structure called a **k-d tree** was one of the early structures used for indexing in multiple dimensions. Each level of a k-d tree partitions the space into two. The partitioning is done along one dimension at the node at the top level of the tree, along another dimension in nodes at the next level, and so on, cycling through the dimensions. The partitioning proceeds in such a way that, at each node, approximately one-half of the points stored in the subtree fall on one side, and one-half fall on the other. Partitioning stops when a node has less than a given maximum number of points. Figure 23.4 shows a set of points in two-dimensional space, and a k-d tree representation of the set of points. Each line corresponds to a node in the tree, and the maximum number of points in a leaf node has been set at 1. Each line in the figure (other than the outside box) corresponds to a node in the k-d tree. The numbering of the lines in the figure indicates the level of the tree at which the corresponding node appears.

The **k-d-B tree** extends the k-d tree to allow multiple child nodes for each internal node, just as a B-tree extends a binary tree, to reduce the height of the tree. k-d-B trees are better suited for secondary storage than k-d trees.



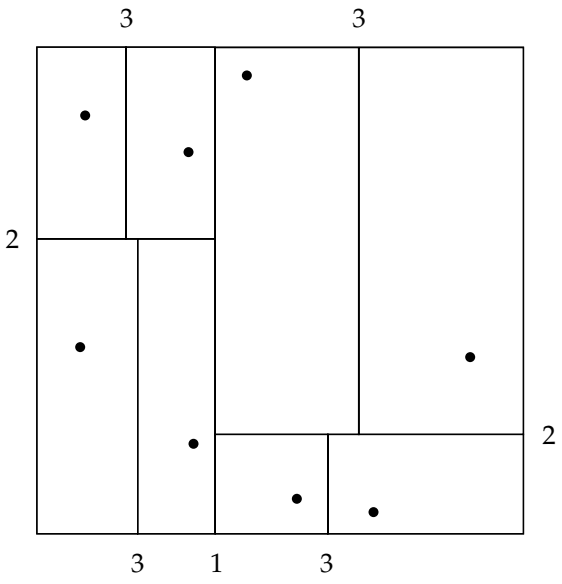



**Figure 23.4** Division of space by a k-d tree.

### 23.3.5.2 Quadtrees

An alternative representation for two-dimensional data is a **quadtree**. An example of the division of space by a quadtree appears in Figure 23.5. The set of points is the same as that in Figure 23.4. Each node of a quadtree is associated with a rectangular region of space. The top node is associated with the entire target space. Each nonleaf node in a quadtree divides its region into four equal-sized quadrants, and correspondingly each such node has four child nodes corresponding to the four quadrants. Leaf nodes have between zero and some fixed maximum number of points. Correspondingly, if the region corresponding to a node has more than the maximum number of points, child nodes are created for that node. In the example in Figure 23.5, the maximum number of points in a leaf node is set to 1.

This type of quadtree is called a **PR quadtree**, to indicate it stores points, and that the division of space is divided based on regions, rather than on the actual set of points stored. We can use **region quadtrees** to store array (raster) information. A node in a region quadtree is a leaf node if all the array values in the region that it covers are the same. Otherwise, it is subdivided further into four children of equal area, and is therefore an internal node. Each node in the region quadtree corresponds to a subarray of values. The subarrays corresponding to leaves either contain just a single array element, or have multiple array elements, all of which have the same value.

Indexing of line segments and polygons presents new problems. There are extensions of k-d trees and quadtrees for this task. However, a line segment or polygon may cross a partitioning line. If it does, it has to be split and represented in each of the subtrees in which its pieces occur. Multiple occurrences of a line segment or polygon can result in inefficiencies in storage, as well as inefficiencies in querying.



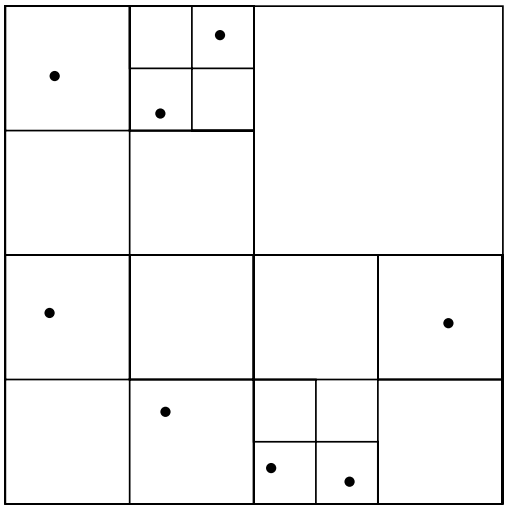


**Figure 23.5** Division of space by a quadtree.

### 23.3.5.3 R-Trees

A storage structure called an **R-tree** is useful for indexing of rectangles and other polygons. An R-tree is a balanced tree structure with the indexed polygons stored in leaf nodes, much like a $B^+$-tree. However, instead of a range of values, a rectangular **bounding box** is associated with each tree node. The bounding box of a leaf node is the smallest rectangle parallel to the axes that contains all objects stored in the leaf node. The bounding box of internal nodes is, similarly, the smallest rectangle parallel to the axes that contains the bounding boxes of its child nodes. The bounding box of a polygon is defined, similarly, as the smallest rectangle parallel to the axes that contains the polygon.

Each internal node stores the bounding boxes of the child nodes along with the pointers to the child nodes. Each leaf node stores the indexed polygons, and may optionally store the bounding boxes of the polygons; the bounding boxes help speed up checks for overlaps of the rectangle with the indexed polygons—if a query rectangle does not overlap with the bounding box of a polygon, it cannot overlap with the polygon either. (If the indexed polygons are rectangles, there is of course no need to store bounding boxes since they are identical to the rectangles.)

Figure 23.6 shows an example of a set of rectangles (drawn with a solid line) and the bounding boxes (drawn with a dashed line) of the nodes of an R-tree for the set of rectangles. Note that the bounding boxes are shown with extra space inside them, to make them stand out pictorially. In reality, the boxes would be smaller and fit tightly on the objects that they contain; that is, each side of a bounding box $B$ would touch at least one of the objects or bounding boxes that are contained in $B$.

The R-tree itself is at the right side of Figure 23.6. The figure refers to the coordinates of bounding box $i$ as $BB_i$ in the figure.



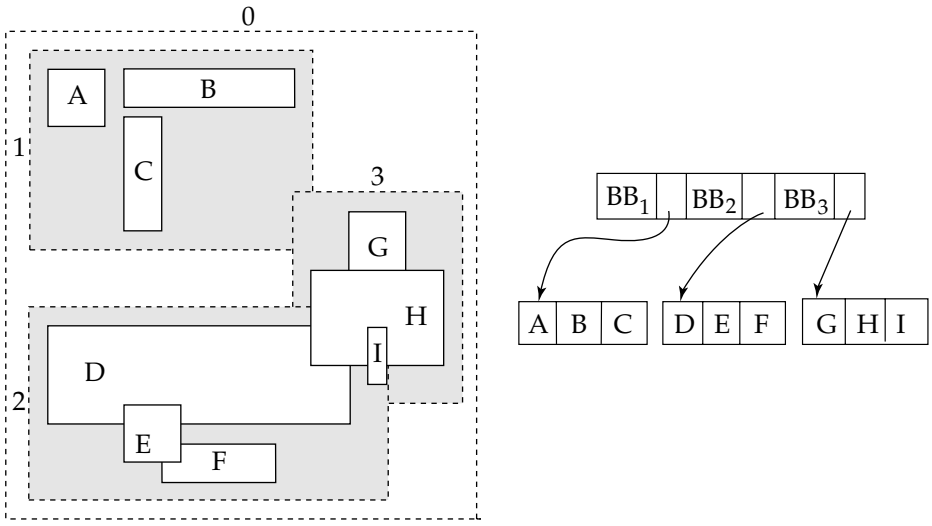



**Figure 23.6** An R-tree.

We shall now see how to implement search, insert, and delete operations on an R-tree.

- **Search**: As the figure shows, the bounding boxes associated with sibling nodes may overlap; in B$^+$-trees, k-d trees, and quadtrees, in contrast, the ranges do not overlap. A search for polygons containing a point therefore has to follow *all* child nodes whose associated bounding boxes contain the point; as a result, multiple paths may have to be searched. Similarly, a query to find all polygons that intersect a given polygon has to go down every node where the associated rectangle intersects the polygon.

- **Insert**: When we insert a polygon into an R-tree, we select a leaf node to hold the polygon. Ideally we should pick a leaf node that has space to hold a new entry, and whose bounding box contains the bounding box of the polygon. However, such a node may not exist; even if it did, finding the node may be very expensive, since it is not possible to find it by a single traversal down from the root. At each internal node we may find multiple children whose bounding boxes contain the bounding box of the polygon, and each of these children needs to be explored. Therefore, as a heuristic, in a traversal from the root, if any of the child nodes has a bounding box containing the bounding box of the polygon, the R-tree algorithm chooses one of them arbitrarily. If none of the children satisfy this condition, the algorithm chooses a child node whose bounding box has the maximum overlap with the bounding box of the polygon for continuing the traversal.

  Once the leaf node has been reached, if the node is already full, the algorithm performs node splitting (and propagates splitting upward if required) in a manner very similar to B$^+$-tree insertion. Just as with B$^+$-tree insertion, the R-tree insertion algorithm ensures that the tree remains balanced. Additionally, it ensures that the bounding boxes of leaf nodes, as well as internal nodes, remain consistent; that is, bounding boxes of leaves contain all the bounding boxes of the polygons stored at the leaf, while the bounding boxes for internal nodes contain all the bounding boxes of the children nodes.

  The main difference of the insertion procedure from the B$^+$-tree insertion procedure lies in how the node is split. In a B$^+$-tree, it is possible to find a value such that half the entries are less than the midpoint and half are greater than the value. This property does not generalize beyond one dimension; that is, for more than one dimension, it is not always possible to split the entries into two sets so that their bounding boxes do not overlap. Instead, as a heuristic, the set of entries $S$ can be split into two disjoint sets $S_1$ and $S_2$ so that the bounding boxes of $S_1$ and $S_2$ have the minimum total area; another heuristic would be to split the entries into two sets $S_1$ and $S_2$ in such a way that $S_1$ and $S_2$ have minimum overlap. The two nodes resulting from the split would contain the entries in $S_1$ and $S_2$ respectively. The cost of finding splits with minimum total area or overlap can itself be large, so cheaper heuristics, such as the *quadratic split* heuristic are used. (The heuristic gets is name from the fact that it takes time quadratic in the number of entries.)

| Workflow application | Typical task | Typical processing entity |
|---|---|---|
| electronic-mail routing | electronic-mail message | mailers |
| loan processing | form processing | humans, application software |
| purchase-order processing | form processing | humans, application software, DBMSs |

**Figure 24.3** Examples of workflows.

performs the tasks may be a person or a software system (for example, a mailer, an application program, or a database-management system).

Figure 24.3 shows examples of workflows. A simple example is that of an electronic-mail system. The delivery of a single mail message may involve several mailer systems that receive and forward the mail message, until the message reaches its destination, where it is stored. Each mailer performs a task—forwarding the mail to the next mailer—and the tasks of multiple mailers may be required to route mail from source to destination. Other terms used in the database and related literature to refer to workflows include **task flow** and **multisystem applications**. Workflow tasks are also sometimes called **steps**.

In general, workflow may involve one or more humans. For instance, consider the processing of a loan. The relevant workflow appears in Figure 24.4. The person who wants a loan fills out a form, which is then checked by a loan officer. An employee who processes loan applications verifies the data in the form, using sources such as credit-reference bureaus. When all the required information has been collected, the loan officer may decide to approve the loan; that decision may then have to be approved by one or more superior officers, after which the loan can be made. Each human here performs a task; in a bank that has not automated the task of loan processing, the coordination of the tasks is typically carried out by passing of the loan application, with attached notes and other information, from one employee to

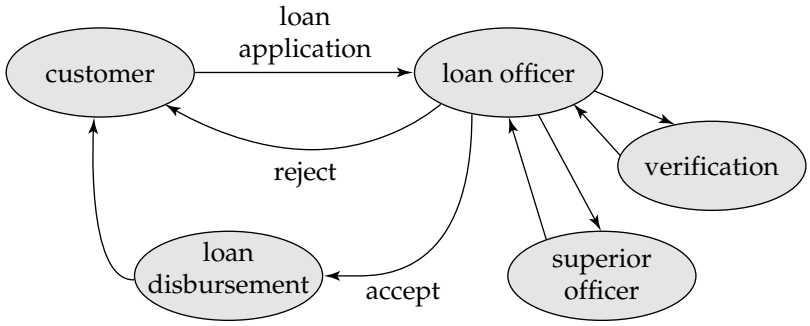



**Figure 24.4** Workflow in loan processing.

message-based workflow systems are particularly useful in networks that may be disconnected for part of the time, such as dial-up networks.

The centralized approach is used in workflow systems where the data are stored in a central database. The scheduler notifies various agents, such as humans or computer programs, that a task has to be carried out, and keeps track of task completion. It is easier to keep track of the state of a workflow with a centralized approach than it is with a fully distributed approach.

The scheduler must guarantee that a workflow will terminate in one of the specified acceptable termination states. Ideally, before attempting to execute a workflow, the scheduler should examine that workflow to check whether the workflow may terminate in a nonacceptable state. If the scheduler cannot guarantee that a workflow will terminate in an acceptable state, it should reject such specifications without attempting to execute the workflow. As an example, let us consider a workflow consisting of two tasks represented by subtransactions $S_1$ and $S_2$, with the failure-atomicity requirements indicating that either both or neither of the subtransactions should be committed. If $S_1$ and $S_2$ do not provide prepared-to-commit states (for a two-phase commit), and further do not have compensating transactions, then it is possible to reach a state where one subtransaction is committed and the other aborted, and there is no way to bring both to the same state. Therefore, such a workflow specification is **unsafe**, and should be rejected.

Such checks such as the one just described may be impossible or impractical to implement in the scheduler; it then becomes the responsibility of the person designing the workflow specification to ensure that the workflows are safe.



### 24.2.4 Recovery of a Workflow

The objective of **workflow recovery** is to enforce the failure atomicity of the workflows. The recovery procedures must make sure that, if a failure occurs in any of the workflow-processing components (including the scheduler), the workflow will eventually reach an acceptable termination state (whether aborted or committed). For example, the scheduler could continue processing after failure and recovery, as though nothing happened, thus providing forward recoverability. Otherwise, the scheduler could abort the whole workflow (that is, reach one of the global abort states). In either case, some subtransactions may need to be committed or even submitted for execution (for example, compensating subtransactions).

We assume that the processing entities involved in the workflow have their own local recovery systems and handle their local failures. To recover the execution-environment context, the failure-recovery routines need to restore the state information of the scheduler at the time of failure, including the information about the execution states of each task. Therefore, the appropriate status information must be logged on stable storage.

We also need to consider the contents of the message queues. When one agent hands off a task to another, the handoff should be carried out exactly once: If the handoff happens twice a task may get executed twice; if the handoff does not occur, the task may get lost. Persistent messaging (Section 19.4.3) provides exactly the features to ensure positive, single handoff.