CHAPTER

5

Physical Database Design and Performance

Learning Objectives

After studying this chapter, you should be able to:

- Concisely define each of the following key terms: **field, data type, denormalization, horizontal partitioning, vertical partitioning, physical file, tablespace, extent, file organization, sequential file organization, indexed file organization, index, secondary key, join index, hashed file organization, hashing algorithm, pointer,** and **hash index table.**
- Describe the physical database design process, its objectives, and its deliverables.
- Choose storage formats for attributes from a logical data model.
- Select an appropriate file organization by balancing various important design factors.
- Describe three important types of file organization.
- Describe the purpose of indexes and the important considerations in selecting attributes to be indexed.
- Translate a relational data model into efficient database structures, including knowing when and how to denormalize the logical data model.

INTRODUCTION

In Chapters 2 through 4, you learned how to describe and model organizational data during the conceptual data modeling and logical database design phases of the database development process. You learned how to use EER notation, the relational data model, and normalization to develop abstractions of organizational data that capture the meaning of data. However, these notations do not explain how data will be processed or stored. The purpose of physical database design is to translate the logical description of data into the technical specifications for storing and retrieving data. The goal is to create a design for storing data that will provide adequate performance and ensure database integrity, security, and recoverability.

Physical database design does not include implementing files and databases (i.e., creating them and loading data into them). Physical database design produces the technical specifications that programmers, database administrators, and others involved in information systems construction will use during the implementation phase, which we discuss in Chapters 6 through 9.

In this chapter, you study the basic steps required to develop an efficient and high-integrity physical database design; security and recoverability are addressed in Chapter 11. We concentrate in this chapter on the design of a single, centralized database. Later, in Chapter 12, you learn about the design of databases that are stored at multiple, distributed sites. In this chapter, you learn how to estimate the amount of data that users will require in the database and determine how data are likely to be used. You learn about choices for storing attribute values and how to select from among these choices to achieve efficiency and data quality. Because of recent U.S. and international regulations (e.g., Sarbanes-Oxley) on financial reporting by organizations, proper controls specified in physical database design are required as a sound foundation for compliance. Hence, we place special emphasis on data quality measures you can implement within the physical design. You will also learn why normalized tables are not always the basis for the best physical data files and how you can denormalize the data to improve the speed of data retrieval. Finally, you learn about the use of indexes, which are important in speeding up the retrieval of data. In essence, you learn in this chapter how to made databases really "hum."

You must carefully perform physical database design, because the decisions made during this stage have a major impact on data accessibility, response times, data quality, security, user friendliness, and similarly important information system design factors. Database administration (described in Chapter 11) plays a major role in physical database design, so we return to some advanced design issues in that chapter.

THE PHYSICAL DATABASE DESIGN PROCESS

To make life a little easier for you, many physical database design decisions are implicit or eliminated when you choose the database management technologies to use with the information system you are designing. Because many organizations have standards for operating systems, database management systems, and data access languages, you must deal only with those choices not implicit in the given technologies. Thus, we will cover only those decisions that you will make most frequently, as well as other selected decisions that may be critical for some types of applications, such an online data capture and retrieval.

The primary goal of physical database design is data processing efficiency. Today, with ever-decreasing costs for computer technology per unit of measure (both speed and space measures), it is typically very important to design a physical database to minimize the time required by users to interact with the information system. Thus, we concentrate on how to make processing of physical files and databases efficient, with less attention on minimizing the use of space.

Designing physical files and databases requires certain information that should have been collected and produced during prior systems development phases. The information needed for physical file and database design includes these requirements:

- Normalized relations, including estimates for the range of the number of rows in each table
- Definitions of each attribute, along with physical specifications such as maximum possible length
- Descriptions of where and when data are used in various ways (entered, retrieved, deleted, and updated, including typical frequencies of these events)
- Expectations or requirements for response time and data security, backup, recovery, retention, and integrity
- Descriptions of the technologies (database management systems) used for implementing the database

Physical database design requires several critical decisions that will affect the integrity and performance of the application system. These key decisions include the following:

- Choosing the storage format (called *data type*) for each attribute from the logical data model. The format and associated parameters are chosen to maximize data integrity and to minimize storage space.

- Giving the database management system guidance regarding how to group attributes from the logical data model into *physical records*. You will discover that although the columns of a relational table as specified in the logical design are a natural definition for the contents of a physical record, this does not always form the foundation for the most desirable grouping of attributes.
- Giving the database management system guidance regarding how to arrange similarly structured records in secondary memory (primarily hard disks), using a structure (called a *file organization*) so that individual and groups of records can be stored, retrieved, and updated rapidly. Consideration must also be given to protecting data and recovering data if errors are found.
- Selecting structures (including *indexes* and the overall *database architecture*) for storing and connecting files to make retrieving related data more efficient.
- Preparing strategies for handling queries against the database that will optimize performance and take advantage of the file organizations and indexes that you have specified. Efficient database structures will be of benefit only if queries and the database management systems that handle those queries are tuned to intelligently use those structures.

Physical Database Design as a Basis for Regulatory Compliance

One of the primary motivations for strong focus on physical database design is that it forms a foundation for compliance with new national and international regulations on financial reporting. Without careful physical design, an organization cannot demonstrate that its data are accurate and well protected. Laws and regulations such as the Sarbanes-Oxley Act (SOX) in the United States and Basel II for international banking are reactions to recent cases of fraud and deception by executives in major corporations and partners in public accounting firms. The purpose of SOX is to protect investors by improving the accuracy and reliability of corporate disclosures made pursuant to the securities laws, and for other purposes. SOX requires that every annual financial report include an internal control report. This is designed to show that not only are the company's financial data accurate, but the company has confidence in them because adequate controls (e.g., database integrity controls) are in place to safeguard financial data.

SOX is the most recent regulation in a stream of efforts to improve financial data reporting. The Committee of Sponsoring Organizations (COSO) of the Treadway Commission is a voluntary private-sector organization dedicated to improving the quality of financial reporting through business ethics, effective internal controls, and corporate governance. COSO was originally formed in 1985 to sponsor the National Commission on Fraudulent Financial Reporting, an independent private sector initiative that studied the causal factors that can lead to fraudulent financial reporting and developed recommendations for public companies and their independent auditors, for the SEC and other regulators, and for educational institutions. The Control Objectives for Information and Related Technology (COBIT) is an open standard published by the IT Governance Institute and the Information Systems Audit and Control Association. It is an IT control framework built in part upon the COSO framework. The IT Infrastructure Library (ITIL), published by the Office of Government Commerce in Great Britain, focuses on IT services and is often used to complement the COBIT framework.

These standards, guidelines, and rules focus on corporate governance, risk assessment, and security and controls of data. Although laws such as SOX and Basel II require comprehensive audits of all procedures that deal with financial data, compliance can be greatly enhanced by a strong foundation of basic data integrity controls. Because such preventive controls are applied consistently and thoroughly, if designed into the database and enforced by the DBMS, field-level data integrity controls can be viewed very positively in compliance audits. Other DBMS features, such as triggers and stored procedures, discussed in Chapter 7, as well as audit trails and activity logs, discussed in Chapter 11, provide even further ways to ensure that only legitimate data values are stored in the database. However, even these control mechanisms are only as good as the underlying field data controls. Further, for full compliance, all data integrity controls

must be thoroughly documented; defining these controls for the DBMS is a form of documentation. Further, changes to these controls must occur through well-documented change control procedures (so that temporary changes cannot be used to bypass well-designed controls).

Data Volume and Usage Analysis

As mentioned previously, data volume and frequency-of-use statistics are important inputs to the physical database design process, particularly in the case of very large-scale database implementations. Thus, you have to maintain a good understanding of the size and usage patterns of the database throughout its life cycle. In this section, we discuss data volume and usage analysis as if it were a one-time static activity, but in practice, you should continuously monitor significant changes in usage and data volumes.

An easy way to show the statistics about data volumes and usage is by adding notation to the EER diagram that represents the final set of normalized relations from logical database design. Figure 5-1 shows the EER diagram (without attributes) for a simple inventory database in Pine Valley Furniture Company. This EER diagram represents the normalized relations constructed during logical database design for the original conceptual data model of this situation depicted in Figure 3-5b.

Both data volume and access frequencies are shown in Figure 5-1. For example, there are 3,000 PARTs in this database. The supertype PART has two subtypes, MANUFACTURED (40 percent of all PARTs are manufactured) and PURCHASED (70 percent are purchased; because some PARTs are of both subtypes, the percentages sum to more than 100 percent). The analysts at Pine Valley estimate that there are typically 150 SUPPLIERs, and Pine Valley receives, on average, 40 SUPPLIES instances from each SUPPLIER, yielding a total of 6,000 SUPPLIES. The dashed arrows represent access frequencies. So, for example, across all applications that use this database, there are on average 20,000 accesses per hour of PART data, and these yield, based on subtype percentages, 14,000 accesses per hour to PURCHASED PART data. There are an additional 6,000 direct accesses to PURCHASED PART data. Of this total of 20,000 accesses to

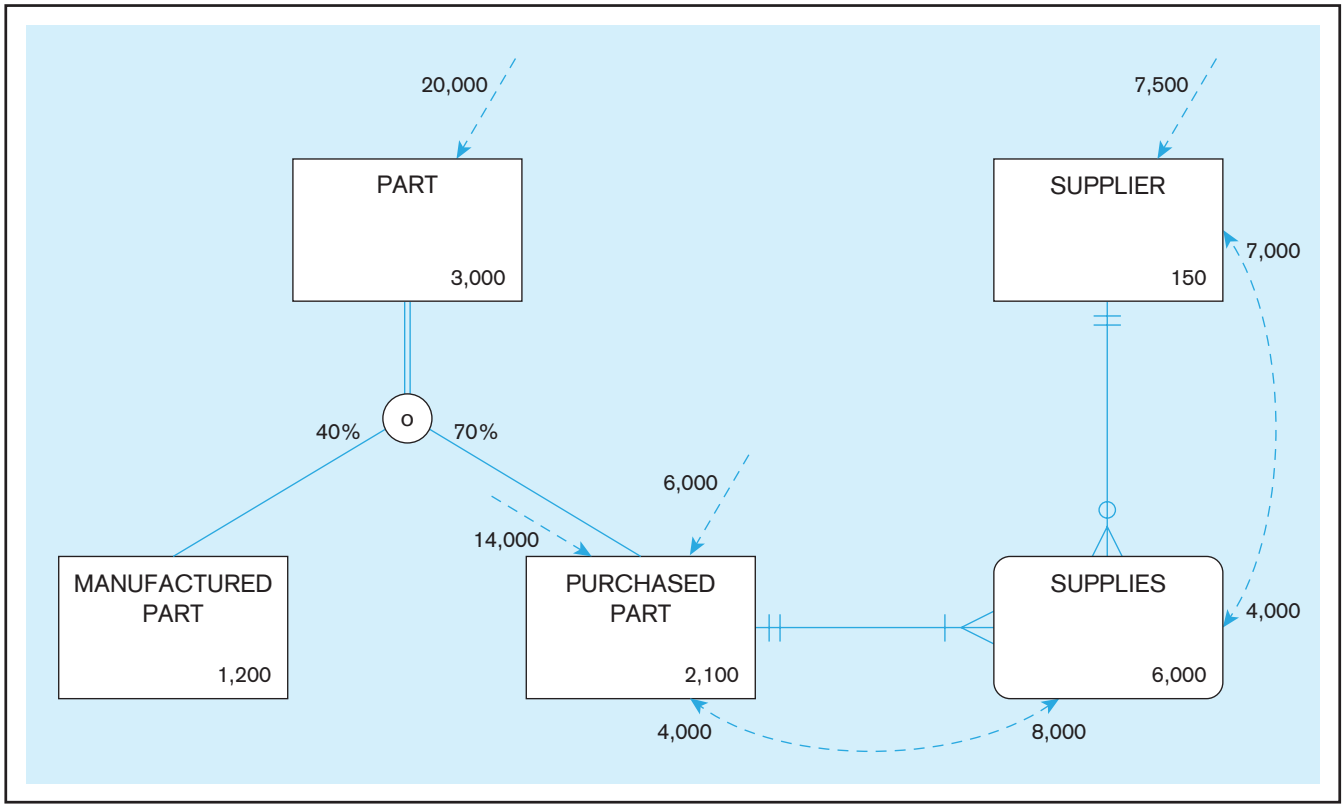


FIGURE 5-1 Composite usage map (Pine Valley Furniture Company)

PURCHASED PART, 8,000 accesses then also require SUPPLIES data and of these 8,000 accesses to SUPPLIES, there are 7,000 subsequent accesses to SUPPLIER data. For online and Web-based applications, usage maps should show the accesses per second. Several usage maps may be needed to show vastly different usage patterns for different times of day. Performance will also be affected by network specifications.

The volume and frequency statistics are generated during the systems analysis phase of the systems development process when systems analysts are studying current and proposed data processing and business activities. The data volume statistics represent the size of the business and should be calculated assuming business growth over at least a several-year period. The access frequencies are estimated from the timing of events, transaction volumes, the number of concurrent users, and reporting and querying activities. Because many databases support ad hoc accesses, and such accesses may change significantly over time, and known database access can peak and dip over a day, week, or month, the access frequencies tend to be less certain and even than the volume statistics. Fortunately, precise numbers are not necessary. What is crucial is the relative size of the numbers, which will suggest where the greatest attention needs to be given during physical database design in order to achieve the best possible performance. For example, in Figure 5-1, notice that

- There are 3,000 PART instances, so if PART has many attributes and some, like description, would be quite long, then the efficient storage of PART might be important.
- For each of the 4,000 times per hour that SUPPLIES is accessed via SUPPLIER, PURCHASED PART is also accessed; thus, the diagram would suggest possibly combining these two co-accessed entities into a database table (or file). This act of combining normalized tables is an example of denormalization, which we discuss later in this chapter.
- There is only a 10 percent overlap between MANUFACTURED and PURCHASED parts, so it might make sense to have two separate tables for these entities and redundantly store data for those parts that are both manufactured and purchased; such planned redundancy is okay if purposeful. Further, there are a total of 20,000 accesses an hour of PURCHASED PART data (14,000 from access to PART and 6,000 independent access of PURCHASED PART) and only 8,000 accesses of MANUFACTURED PART per hour. Thus, it might make sense to organize tables for MANUFACTURED and PURCHASED PART data differently due to the significantly different access volumes.

It can be helpful for subsequent physical database design steps if you can also explain the nature of the access for the access paths shown by the dashed lines. For example, it can be helpful to know that of the 20,000 accesses to PART data, 15,000 ask for a part or a set of parts based on the primary key, PartNo (e.g., access a part with a particular number); the other 5,000 accesses qualify part data for access by the value of QtyOnHand. (These specifics are not shown in Figure 5-1.) This more precise description can help in selecting indexes, one of the major topics we discuss later in this chapter. It might also be helpful to know whether an access results in data creation, retrieval, update, or deletion. Such a refined description of access frequencies can be handled by additional notation on a diagram such as in Figure 5-1, or by text and tables kept in other documentation.

DESIGNING FIELDS

A **field** is the smallest unit of application data recognized by system software, such as a programming language or database management system. A field corresponds to a simple attribute in the logical data model, and so in the case of a composite attribute, a field represents a single component.

Field
The smallest unit of application data recognized by system software.

The basic decisions you must make in specifying each field concern the type of data (or storage type) used to represent values of this field, data integrity controls built into the database, and the mechanisms that the DBMS uses to handle missing values for the field. Other field specifications, such as display format, also must be made as part of the total specification of the information system, but we will not be concerned here with those specifications that are often handled by applications rather than the DBMS.

Choosing Data Types

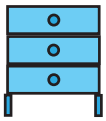
Data type
A detailed coding scheme recognized by system software, such as a DBMS, for representing organizational data.

A **data type** is a detailed coding scheme recognized by system software, such as a DBMS, for representing organizational data. The bit pattern of the coding scheme is usually transparent to you, but the space to store data and the speed required to access data are of consequence in physical database design. The specific DBMS you will use will dictate which choices are available to you. For example, Table 5-1 lists some of the data types available in the Oracle 11g DBMS, a typical DBMS that uses the SQL data definition and manipulation language. Additional data types might be available for currency, voice, image, and user defined for some DBMSs.

Selecting a data type involves four objectives that will have different relative levels of importance for different applications:

1. Represent all possible values.
2. Improve data integrity.
3. Support all data manipulations.
4. Minimize storage space.

An optimal data type for a field can, in minimal space, represent every possible value (while eliminating illegal values) for the associated attribute and can support the required data manipulation (e.g., numeric data types for arithmetic operations and character data types for string manipulation). Any attribute domain constraints from the conceptual data model are helpful in selecting a good data type for that attribute. Achieving these four objectives can be subtle. For example, consider a DBMS for which a data type has a maximum width of 2 bytes. Suppose this data type is sufficient to represent a QuantitySold field. When QuantitySold fields are summed, the sum may require a number larger than 2 bytes. If the DBMS uses the field's data type for results of any mathematics on that field, the 2-byte length will not work. Some data types have special manipulation capabilities; for example, only the DATE data type allows true date arithmetic.

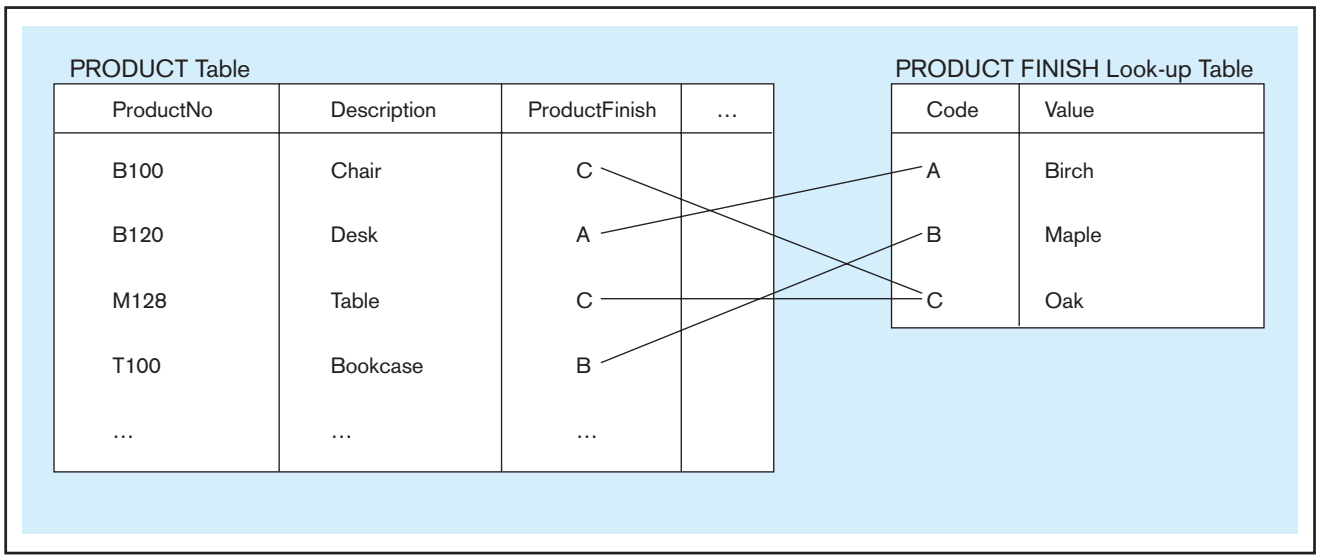


CODING TECHNIQUES Some attributes have a sparse set of values or are so large that, given data volumes, considerable storage space will be consumed. A field with a limited number of possible values can be translated into a code that requires less space. Consider the example of the ProductFinish field illustrated in Figure 5-2. Products at

TABLE 5-1 Commonly Used Data Types in Oracle 11g

Data Type	Description
VARCHAR2	Variable-length character data with a maximum length of 4,000 characters; you must enter a maximum field length (e.g., VARCHAR2(30) specifies a field with a maximum length of 30 characters). A value less than 30 characters will consume only the required space.
CHAR	Fixed-length character data with a maximum length of 2,000 characters; default length is 1 character (e.g., CHAR(5) specifies a field with a fixed length of 5 characters, capable of holding a value from 0 to 5 characters long).
CLOB	Character large object, capable of storing up to 4 gigabytes of one variable-length character data field (e.g., to hold a medical instruction or a customer comment).
NUMBER	Positive or negative number in the range 10^{-130} to 10^{126}; can specify the precision (total number of digits to the left and right of the decimal point) and the scale (the number of digits to the right of the decimal point) (e.g., NUMBER(5) specifies an integer field with a maximum of 5 digits, and NUMBER(5,2) specifies a field with no more than 5 digits and exactly 2 digits to the right of the decimal point).
INTEGER	Positive or negative integer with up to 38 digits (same as SMALL INT).
DATE	Any date from January 1, 4712 B.C., to December 31, 9999 A.D.; DATE stores the century, year, month, day, hour, minute, and second.
BLOB	Binary large object, capable of storing up to 4 gigabytes of binary data (e.g., a photograph or sound clip).

FIGURE 5-2 Example of a code lookup table (Pine Valley Furniture Company)



Pine Valley Furniture come in only a limited number of woods: Birch, Maple, and Oak. By creating a code or translation table, each ProductFinish field value can be replaced by a code, a cross-reference to the lookup table, similar to a foreign key. This will decrease the amount of space for the ProductFinish field and hence for the PRODUCT file. There will be additional space for the PRODUCT FINISH lookup table, and when the ProductFinish field value is needed, an extra access (called a join) to this lookup table will be required. If the ProductFinish field is infrequently used or if the number of distinct ProductFinish values is very large, the relative advantages of coding may outweigh the costs. Note that the code table would not appear in the conceptual or logical model. The code table is a physical construct to achieve data processing performance improvements, not a set of data with business value.

Controlling Data Integrity For many DBMSs, data integrity controls (i.e., controls on the possible value a field can assume) can be built into the physical structure of the fields and controls enforced by the DBMS on those fields. The data type enforces one form of data integrity control because it may limit the type of data (numeric or character) and the length of a field value. The following are some other typical integrity controls that a DBMS may support:

- ***Default value*** A default value is the value a field will assume unless a user enters an explicit value for an instance of that field. Assigning a default value to a field can reduce data entry time because entry of a value can be skipped. It can also help to reduce data entry errors for the most common value.
- ***Range control*** A range control limits the set of permissible values a field may assume. The range may be a numeric lower-to-upper bound or a set of specific values. Range controls must be used with caution because the limits of the range may change over time. A combination of range controls and coding led to the year 2000 problem that many organizations faced, in which a field for year was represented by only the numbers 00 to 99. It is better to implement any range controls through a DBMS because range controls in applications may be inconsistently enforced. It is also more difficult to find and change them in applications than in a DBMS.
- ***Null value control*** A null value was defined in Chapter 4 as an empty value. Each primary key must have an integrity control that prohibits a null value. Any other required field may also have a null value control placed on it if that is the policy of the organization. For example, a university may prohibit adding a course to its database unless that course has a title as well as a value of the primary key, CourseID. Many fields legitimately may have a null value, so this control should be used only when truly required by business rules.

- ***Referential integrity*** The term *referential integrity* was defined in Chapter 4. Referential integrity on a field is a form of range control in which the value of that field must exist as the value in some field in another row of the same or (most commonly) a different table. That is, the range of legitimate values comes from the dynamic contents of a field in a database table, not from some pre-specified set of values. Note that referential integrity guarantees that only some existing cross-referencing value is used, not that it is the correct one. A coded field will have referential integrity with the primary key of the associated lookup table.

HANDLING MISSING DATA When a field may be null, simply entering no value may be sufficient. For example, suppose a customer zip code field is null and a report summarizes total sales by month and zip code. How should sales to customers with unknown zip codes be handled? Two options for handling or preventing missing data have already been mentioned: using a default value and not permitting missing (null) values. Missing data are inevitable. According to Babad and Hoffer (1984), the following are some other possible methods for handling missing data:

- Substitute an estimate of the missing value. For example, for a missing sales value when computing monthly product sales, use a formula involving the mean of the existing monthly sales values for that product indexed by total sales for that month across all products. Such estimates must be marked so that users know that these are not actual values.
- Track missing data so that special reports and other system elements cause people to resolve unknown values quickly. This can be done by setting up a trigger in the database definition. A trigger is a routine that will automatically execute when some event occurs or time period passes. One trigger could log the missing entry to a file when a null or other missing value is stored, and another trigger could run periodically to create a report of the contents of this log file.
- Perform sensitivity testing so that missing data are ignored unless knowing a value might significantly change results (e.g., if total monthly sales for a particular salesperson are almost over a threshold that would make a difference in that person's compensation). This is the most complex of the methods mentioned and hence requires the most sophisticated programming. Such routines for handling missing data may be written in application programs. All relevant modern DBMSs now have more sophisticated programming capabilities, such as case expressions, user-defined functions, and triggers, so that such logic can be available in the database for all users without application-specific programming.

DENORMALIZING AND PARTITIONING DATA

Modern database management systems have an increasingly important role in determining how the data are actually stored on the storage media. The efficiency of database processing is, however, significantly affected by how the logical relations are structured as database tables. The purpose of this section is to discuss denormalization as a mechanism that is often used to improve efficient processing of data and quick access to stored data. It first describes the best-known denormalization approach: combining several logical tables into one physical table to avoid the need to bring related data back together when they are retrieved from the database. Then the section will discuss another form of denormalization called *partitioning*, which also leads to differences between the logical data model and the physical tables, but in this case one relation is implemented as multiple tables.

Denormalization

With the rapid decline in the costs of secondary storage per unit of data, the efficient use of storage space (reducing redundancy), while still a relevant consideration, has become less important than it has been in the past. In most cases, the primary goal of physical record design—efficient data processing—dominates the design process.

In other words, speed, not style, matters. As in your dorm room, as long as you can find your favorite sweat shirt when you need it, it doesn't matter how tidy the room looks. (We won't tell your Mom.)

Efficient processing of data, just like efficient accessing of books in a library, depends on how close together related data (books or indexes) are. Often all the attributes that appear within a relation are not used together, and data from different relations are needed together to answer a query or produce a report. Thus, although normalized relations solve data maintenance anomalies and minimize redundancies (and storage space), normalized relations, if implemented one for one as physical records, may not yield efficient data processing.

A fully normalized database usually creates a large number of tables. For a frequently used query that requires data from multiple, related tables, the DBMS can spend considerable computer resources each time the query is submitted in matching up (called *joining*) related rows from each table required to build the query result. Because this joining work is so time-consuming, the processing performance difference between totally normalized and partially normalized databases can be dramatic. Inmon (1988) reports on a study to quantify fully and partially normalized databases. A fully normalized database contained eight tables with about 50,000 rows each; another partially normalized database had four tables with roughly 25,000 rows each; and yet another partially normalized database had two tables. The result showed that the less-than-fully normalized databases could be as much as an order of magnitude faster than the fully normalized one. Although such results depend greatly on the database and the type of processing against it, these results suggest that you should carefully consider whether the physical structure should exactly match the normalized relations for a database.

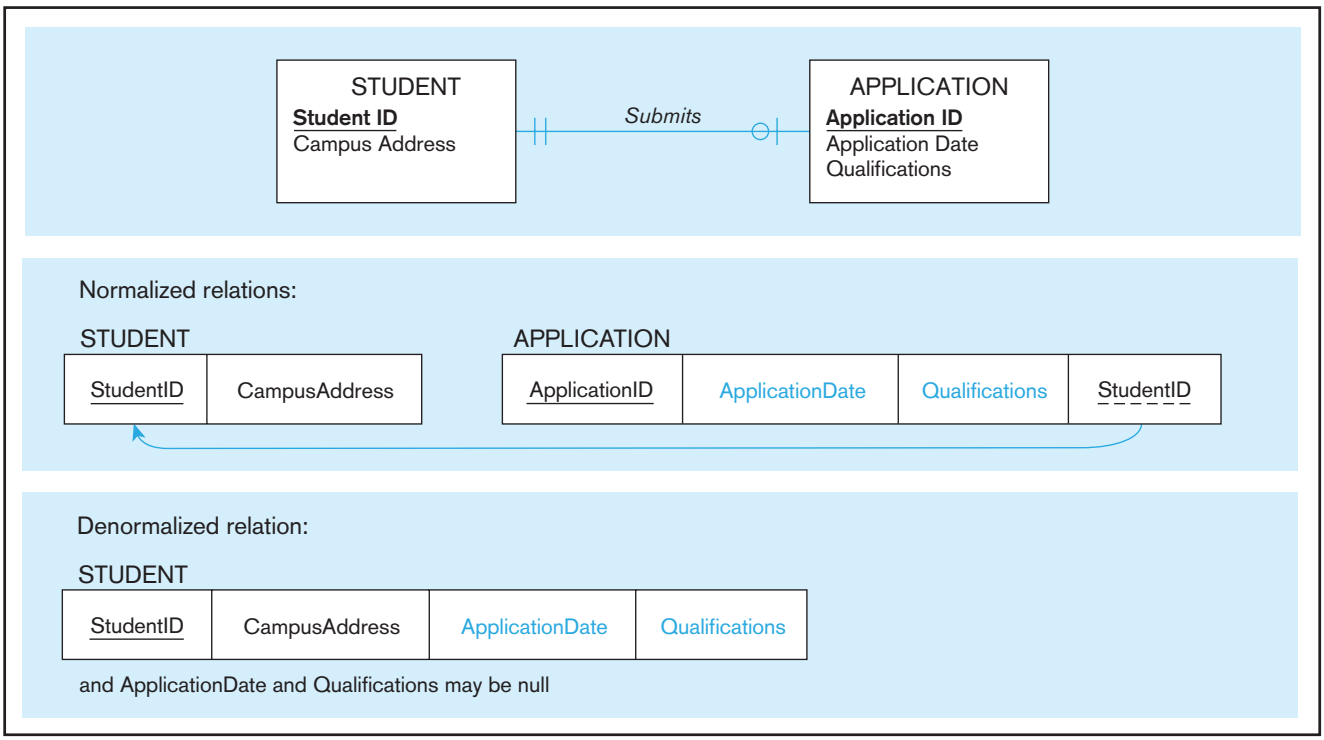
Denormalization is the process of transforming normalized relations into non-normalized physical record specifications. We will review various forms of, reasons for, and cautions about denormalization in this section. In general, denormalization may partition a relation into several physical records, may combine attributes from several relations together into one physical record, or may do a combination of both.

Denormalization
The process of transforming normalized relations into non-normalized physical record specifications.

OPPORTUNITIES FOR AND TYPES OF DENORMALIZATION Rogers (1989) introduces several common denormalization opportunities (Figures 5-3 through Figure 5-5 show examples of normalized and denormalized relations for each of these three situations):

1. ***Two entities with a one-to-one relationship*** Even if one of the entities is an optional participant, if the matching entity exists most of the time, then it may be wise to combine these two relations into one record definition (especially if the access frequency between these two entity types is high). Figure 5-3 shows student data with optional data from a standard scholarship application a student may complete. In this case, one record could be formed with four fields from the STUDENT and SCHOLARSHIP APPLICATION normalized relations (assuming that ApplicationID is no longer needed). (Note: In this case, fields from the optional entity must have null values allowed.)
2. ***A many-to-many relationship (associative entity) with nonkey attributes*** Rather than join three files to extract data from the two basic entities in the relationship, it may be advisable to combine attributes from one of the entities into the record representing the many-to-many relationship, thus avoiding one of the join operations. Again, this would be most advantageous if this joining occurs frequently. Figure 5-4 shows price quotes for different items from different vendors. In this case, fields from ITEM and PRICE QUOTE relations might be combined into one record to avoid having to join all three tables together. (Note: This may create considerable duplication of data; in the example, the ITEM fields, such as Description, would repeat for each price quote. This would necessitate excessive updating if duplicated data changed. Careful analysis of a composite usage map to study access frequencies and the number of occurrences of PRICE QUOTE per associated VENDOR or ITEM would be essential to understand the consequences of such denormalization.)

FIGURE 5-3 **A possible denormalization situation: two entities with a one-to-one relationship**



(*Note:* We assume that ApplicationID is not necessary when all fields are stored in one record, but this field can be included if it is required application data.)

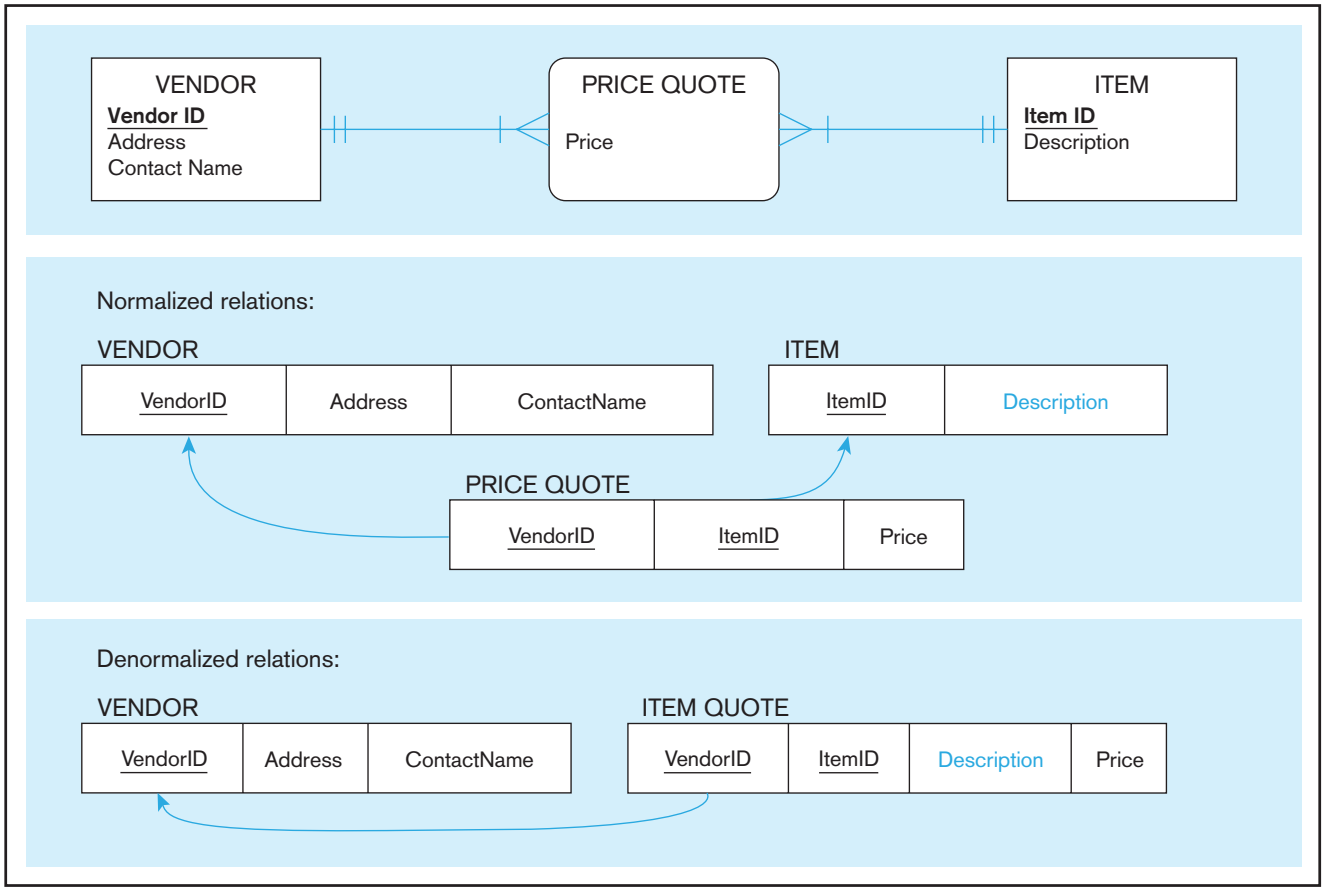
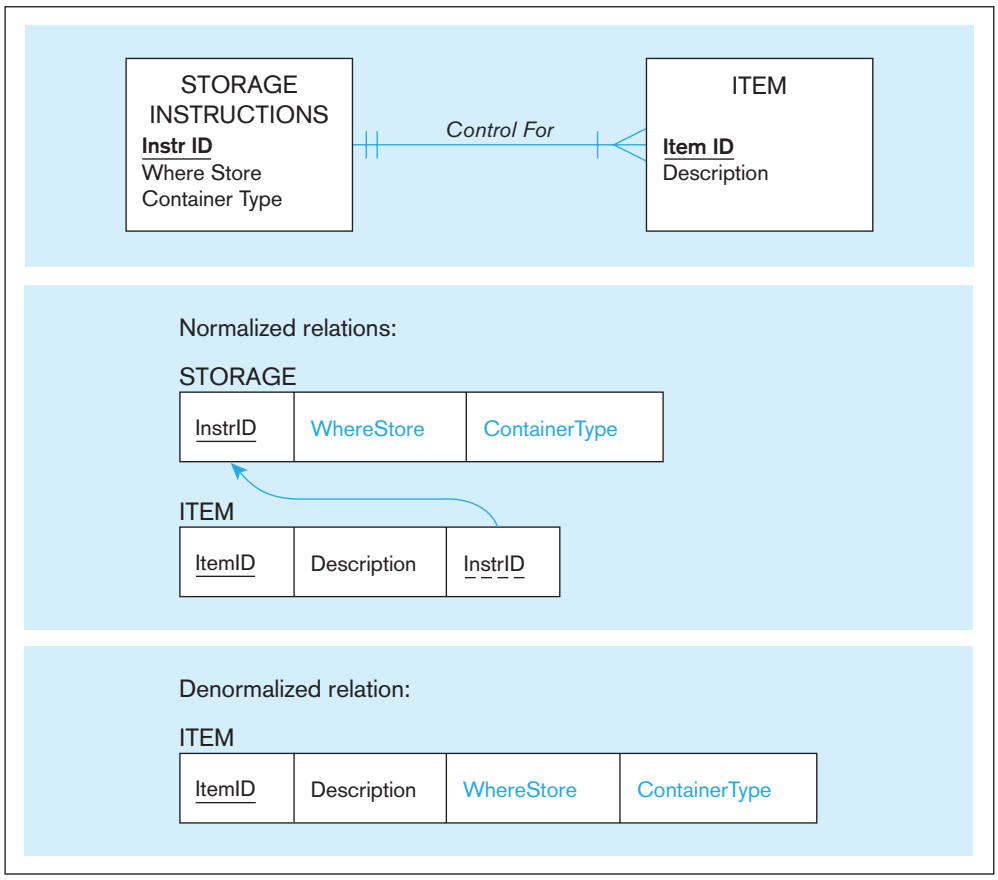


FIGURE 5-4 **A possible denormalization situation: a many-to-many relationship with nonkey attributes**

FIGURE 5-5 **A possible denormalization situation: reference data**



3. ***Reference data*** Reference data exist in an entity on the one side of a one-to-many relationship, and this entity participates in no other database relationships. You should seriously consider merging the two entities in this situation into one record definition when there are few instances of the entity on the many side for each entity instance on the one side. See Figure 5-5, in which several ITEMs have the same STORAGE INSTRUCTIONS, and STORAGE INSTRUCTIONS relates only to ITEMs. In this case, the storage instructions data could be stored in the ITEM record to create, of course, redundancy and potential for extra data maintenance. (InstrID is no longer needed.)

DENORMALIZE WITH CAUTION Denormalization has it critics. As Finkelstein (1988) points out, denormalization can increase the chance of errors and inconsistencies (caused by reintroducing anomalies into the database) and can force the reprogramming of systems if business rules change. For example, redundant copies of the same data caused by a violation of second normal form are often not updated in a synchronized way. And, if they are, extra programming is required to ensure that all copies of exactly the same business data are updated together. Further, denormalization optimizes certain data processing at the expense of other data processing, so if the frequencies of different processing activities change, the benefits of denormalization may no longer exist. Denormalization almost always also leads to more storage space for raw data and maybe more space for database overhead (e.g., indexes). Thus, denormalization should be an explicit act to gain significant processing speed when other physical design actions are not sufficient to achieve processing expectations.

Pascal (2002a, 2002b) passionately reports of the many dangers of denormalization. The motivation for denormalization is that a normalized database often creates many tables, and joining tables slows database processing. Pascal argues that this is not necessarily true, so the motivation for denormalization may be without merit in some

cases. Overall, performance does not depend solely on the number of tables accessed but rather also on how the tables are organized in the database (what we later call *file organizations* and *clustering*), the proper design and implementation of queries, and the query optimization capabilities of the DBMS. Thus, to avoid problems associated with the data anomalies in denormalized databases, Pascal recommends first attempting to use these other means to achieve the necessary performance. This often will be sufficient, but in cases when further steps are needed, you must understand the opportunities for applying denormalization.

Hoberman (2002) has written a very useful two-part "denormalization survival guide," which summarizes the major factors (those outlined previously and a few others) in deciding whether to denormalize.

Partitioning

The opportunities just listed all deal with combining tables to avoid doing joins. Another form of denormalization involves the creation of more tables by partitioning a relation into multiple physical tables. Either horizontal or vertical partitioning, or a combination, is possible. **Horizontal partitioning** implements a logical relation as multiple physical tables by placing different rows into different tables, based on common column values. (In a library setting, horizontal partitioning is similar to placing the business journals in a business library, the science books in a science library, and so on.) Each table created from the partitioning has the same columns. For example, a customer relation could be broken into four regional customer tables based on the value of a column Region.

Horizontal partitioning
Distribution of the rows of a logical relation into several separate tables.

Horizontal partitioning makes sense when different categories of rows of a table are processed separately (e.g., for the Customer table just mentioned, if a high percentage of the data processing needs to work with only one region at a time). Two common methods of horizontal partitioning are to partition on (1) a single column value (e.g., CustomerRegion) and (2) date (because date is often a qualifier in queries, so just the needed partitions can be quickly found). (See Bieniek, 2006, for a guide to table partitioning.) Horizontal partitioning can also make maintenance of a table more efficient because fragmenting and rebuilding can be isolated to single partitions as storage space needs to be reorganized. Horizontal partitioning can also be more secure because file-level security can be used to prohibit users from seeing certain rows of data. Also, each partitioned table can be organized differently, appropriately for the way it is individually used. It is likely also faster to recover one of the partitioned files than one file with all the rows. In addition, taking one of the partitioned files out of service because it was damaged or so it can be recovered still allows processing against the other partitioned files to continue. Finally, each of the partitioned files can be placed on a separate disk drive to reduce contention for the same drive and hence improve query and maintenance performance across the database. These advantages of horizontal partitioning (actually, all forms of partitioning), along with the disadvantages, are summarized in Table 5-2.

Note that horizontal partitioning is very similar to creating a supertype/subtype relationship because different types of the entity (where the subtype discriminator is the field used for segregating rows) are involved in different relationships, hence different processing. In fact, when you have a supertype/subtype relationship, you need to decide whether you will create separate tables for each subtype or combine them in various combinations. Combining makes sense when all subtypes are used about the same way, whereas partitioning the supertype entity into multiple files makes sense when the subtypes are handled differently in transactions, queries, and reports. When a relation is partitioned horizontally, the whole set of rows can be reconstructed by using the SQL UNION operator (described in Chapter 6). Thus, for example, all customer data can be viewed together when desired.

The Oracle DBMS supports several forms of horizontal partitioning, designed in particular to deal with very large tables (Brobst et al., 1999). A table is partitioned when it is defined to the DBMS using the SQL data definition language (you will learn about the CREATE TABLE command in Chapter 6); that is, in Oracle, there is one table with

TABLE 5-2 Advantages and Disadvantages of Data Partitioning

Advantages of Partitioning
1. *Efficiency*: Data queried together are stored close to one another and separate from data not used together. Data maintenance is isolated in smaller partitions.
2. *Local optimization*: Each partition of data can be stored to optimize performance for its own use.
3. *Security*: Data not relevant to one group of users can be segregated from data those users are allowed to use.
4. *Recovery and uptime*: Smaller files take less time to back up and recover, and other files are still accessible if one file is damaged, so the effects of damage are isolated.
5. *Load balancing*: Files can be allocated to different storage areas (disks or other media), which minimizes contention for access to the same storage area or even allows for parallel access to the different areas.
Disadvantages of Partitioning
1. *Inconsistent access speed*: Different partitions may have different access speeds, thus confusing users. Also, when data must be combined across partitions, users may have to deal with significantly slower response times than in a non-partitioned approach.
2. *Complexity*: Partitioning is usually not transparent to programmers, who will have to write more complex programs when combining data across partitions.
3. *Extra space and update time*: Data may be duplicated across the partitions, taking extra storage space compared to storing all the data in normalized files. Updates that affect data in multiple partitions can take more time than if one file were used.

several partitions rather than separate tables per se. Oracle 11g has three data distribution methods as basic partitioning approaches:

1. ***Range partitioning,*** in which each partition is defined by a range of values (lower and upper key value limits) for one or more columns of the normalized table. A table row is inserted in the proper partition, based on its initial values for the range fields. Because partition key values may follow patterns, each partition may hold quite a different number of rows. A partition key may be generated by the database designer to create a more balanced distribution of rows. A row may be restricted from moving between partitions when key values are updated.
2. ***Hash partitioning,*** in which data are evenly spread across partitions independent of any partition key value. Hash partitioning overcomes the uneven distribution of rows that is possible with range partitioning. It works well if the goal is to distribute data evenly across devices.
3. ***List partitioning,*** in which the partitions are defined based on predefined lists of values of the partitioning key. For example, in a table partitioned based on the value of the column State, one partition might include rows that have the value "CT," "ME," "MA," "NH," "RI," or "VT," and another partition rows that have the value "NJ" or "NY".

If a more sophisticated form of partitioning is needed, Oracle 11g also offers composite partitioning, which combines aspects of two of the three single-level partitioning approaches.

Partitions are in many cases transparent to the database user. (You need to refer to a partition only if you want to force the query processor to look at one or more partitions.) The part of the DBMS that optimizes the processing of a query will look at the definition of partitions for a table involved in a query and will automatically decide whether certain partitions can be eliminated when retrieving the data needed to form the query results, which can drastically improve query processing performance.

For example, suppose a transaction date is used to define partitions in range partitioning. A query asking for only recent transactions can be more quickly processed by looking at only the one or few partitions with the most recent transactions rather than

scanning the database or even using indexes to find rows in the desired range from a non-partitioned table. A partition on date also isolates insertions of new rows to one partition, which may reduce the overhead of database maintenance, and dropping "old" transactions will require simply dropping a partition. Indexes can still be used with a partitioned table and can improve performance even more than partitioning alone. See Brobst et al. (1999) for more details on the pros and cons of using dates for range partitioning.

In hash partitioning, rows are more evenly spread across the partitions. If partitions are placed in different storage areas that can be processed in parallel, then query performance will improve noticeably compared to when all the data have to be accessed sequentially in one storage area for the whole table. As with range partitioning, the existence of partitions typically is transparent to a programmer of a query.

Vertical partitioning
Distribution of the columns of a logical relation into several separate physical tables.

Vertical partitioning distributes the columns of a logical relation into separate tables, repeating the primary key in each of the tables. An example of vertical partitioning would be breaking apart a PART relation by placing the part number along with accounting-related part data into one record specification, the part number along with engineering-related part data into another record specification, and the part number along with sales-related part data into yet another record specification. The advantages and disadvantages of vertical partitioning are similar to those for horizontal partitioning. When, for example, accounting-, engineering-, and sales-related part data need to be used together, these tables can be joined. Thus, neither horizontal nor vertical partitioning prohibits the ability to treat the original relation as a whole.

Combinations of horizontal and vertical partitioning are also possible. This form of denormalization—record partitioning—is especially common for a database whose files are distributed across multiple computers. Thus, you study this topic again in Chapter 12.

A single physical table can be logically partitioned or several tables can be logically combined by using the concept of a user view, which will be demonstrated in Chapter 6. With a user view, users can be given the impression that the database contains tables other than what are physically defined; you can create these logical tables through horizontal or vertical partitioning or other forms of denormalization. However, the purpose of any form of user view, including logical partitioning via views, is to simplify query writing and to create a more secure database, not to improve query performance. One form of a user view available in Oracle is called a partition view. With a partition view, physically separate tables with similar structures can be logically combined into one table using the SQL UNION operator. There are limitations to this form of partitioning. First, because there are actually multiple separate physical tables, there cannot be any global index on all the combined rows. Second, each physical table must be separately managed, so data maintenance is more complex (e.g., a new row must be inserted into a specific table). Third, the query optimizer has fewer options with a partition view than with partitions of a single table for creating the most efficient query processing plan.

The final form of denormalization we introduce is data replication. With data replication, the same data are purposely stored in multiple places in the database. For example, consider again Figure 5-1. You learned earlier in this section that relations can be denormalized by combining data from an associative entity with data from one of the simple entities with which it is associated. So, in Figure 5-1, SUPPLIES data might be stored with PURCHASED PART data in one expanded PURCHASED PART physical record specification. With data duplication, the same SUPPLIES data might also be stored with its associated SUPPLIER data in another expanded SUPPLIER physical record specification. With this data duplication, once either a SUPPLIER or PURCHASED PART record is retrieved, the related SUPPLIES data will also be available without any further access to secondary memory. This improved speed is worthwhile only if SUPPLIES data are frequently accessed with SUPPLIER and with PURCHASED PART data and if the costs for extra secondary storage and data maintenance are not great.

DESIGNING PHYSICAL DATABASE FILES

Physical file
A named portion of secondary memory (such as a hard disk) allocated for the purpose of storing physical records.

A **physical file** is a named portion of secondary memory (such as a magnetic tape or hard disk) allocated for the purpose of storing physical records. Some computer operating systems allow a physical file to be split into separate pieces, sometimes called

extents. In subsequent sections, we will assume that a physical file is not split and that each record in a file has the same structure. That is, subsequent sections address how to store and link relational table rows from a single database in physical storage space. In order to optimize the performance of the database processing, the person who administers a database, the database administrator, often needs to know extensive details about how the database management system manages physical storage space. This knowledge is very DBMS specific, but the principles described in subsequent sections are the foundation for the physical data structures used by most relational DBMSs.

Most database management systems store many different kinds of data in one operating system file. By an *operating system file* we mean a named file that would appear on a disk directory listing (e.g., a listing of the files in a folder on the C: drive of your personal computer). For example, an important logical structure for storage space in Oracle is a tablespace. A **tablespace** is a named logical storage unit in which data from one or more database tables, views, or other database objects may be stored. An instance of Oracle 11g includes many tablespaces—for example, two (SYSTEM and SYSAUX) for system data (data dictionary or data about data), one (TEMP) for temporary work space, one (UNDOTBS1) for undo operations, and one or several to hold user business data. A tablespace consists of one or several physical operating system files. Thus, Oracle has responsibility for managing the storage of data inside a tablespace, whereas the operating system has many responsibilities for managing a tablespace, but they are all related to its responsibilities related to the management of operating system files (e.g., handling file-level security, allocating space, and responding to disk read and write errors).

Tablespace
A named logical storage unit in which data from one or more database tables, views, or other database objects may be stored.

Because an instance of Oracle usually supports many databases for many users, a database administrator usually will create many user tablespaces, which helps to achieve database security because the administrator can give each user selected rights to access each tablespace. Each tablespace consists of logical units called *segments* (consisting of one table, index, or partition), which, in turn, are divided into **extents**. These, finally, consist of a number of contiguous *data blocks*, which are the smallest unit of storage. Each table, index, or other so-called schema object belongs to a single tablespace, but a tablespace may contain (and typically contains) one or more tables, indexes, and other schema objects. Physically, each tablespace can be stored in one or multiple data files, but each data file is associated with only one tablespace and only one database.

Extent
A contiguous section of disk storage space.

Modern database management systems have an increasingly active role in managing the use of the physical devices and files on them; for example, the allocation of schema objects (e.g., tables and indexes) to data files is typically fully controlled by the DBMS. A database administrator does, however, have the ability to manage the disk space allocated to tablespaces and a number of parameters related to the way free space is managed within a database. Because this is not a text on Oracle, we do not cover specific details on managing tablespaces; however, the general principles of physical database design apply to the design and management of Oracle tablespaces as they do to whatever the physical storage unit is for any database management system. Figure 5-6 is an EER model that shows the relationships between various physical and logical database terms related to physical database design in an Oracle environment.

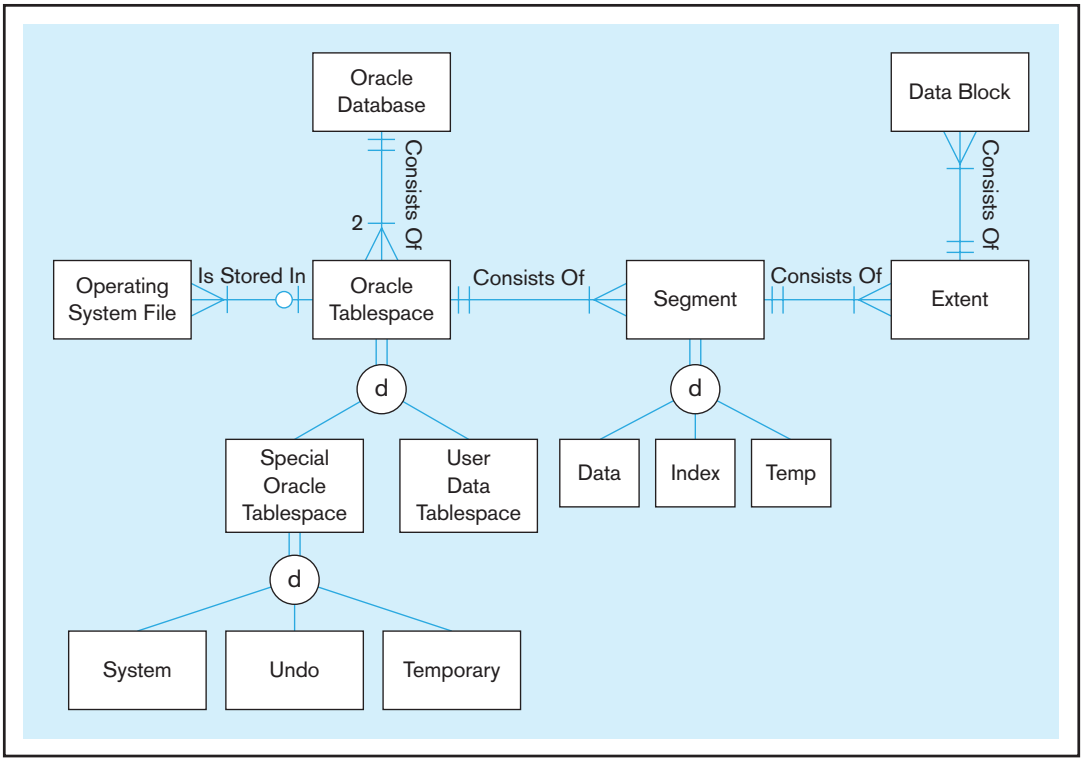
File Organizations

A **file organization** is a technique for physically arranging the records of a file on secondary storage devices. With modern relational DBMSs, you do not have to design file organizations, but you may be allowed to select an organization and its parameters for a table or physical file. In choosing a file organization for a particular file in a database, you should consider seven important factors:

File organization
A technique for physically arranging the records of a file on secondary storage devices.

1. Fast data retrieval
2. High throughput for processing data input and maintenance transactions
3. Efficient use of storage space
4. Protection from failures or data loss
5. Minimizing need for reorganization
6. Accommodating growth
7. Security from unauthorized use

FIGURE 5-6 DBMS terminology in an Oracle 11g environment



Often these objectives are in conflict, and you must select a file organization that provides a reasonable balance among the criteria within resources available.

In this chapter, we consider the following families of basic file organizations: sequential, indexed, and hashed. Figure 5-7 illustrates each of these organizations, with the nicknames of some university sports teams.

Sequential file organization
The storage of records in a file in sequence according to a primary key value.

SEQUENTIAL FILE ORGANIZATIONS In a **sequential file organization**, the records in the file are stored in sequence according to a primary key value (see Figure 5-7a). To locate a particular record, a program must normally scan the file from the beginning until the desired record is located. A common example of a sequential file is the alphabetical list of persons in the white pages of a telephone directory (ignoring any index that may be included with the directory). A comparison of the capabilities of sequential files with the other two types of files appears later in Table 5-3. Because of their inflexibility, sequential files are not used in a database but may be used for files that back up data from a database.

Indexed file organization
The storage of records either sequentially or nonsequentially with an index that allows software to locate individual records.

Index
A table or other data structure used to determine in a file the location of records that satisfy some condition.

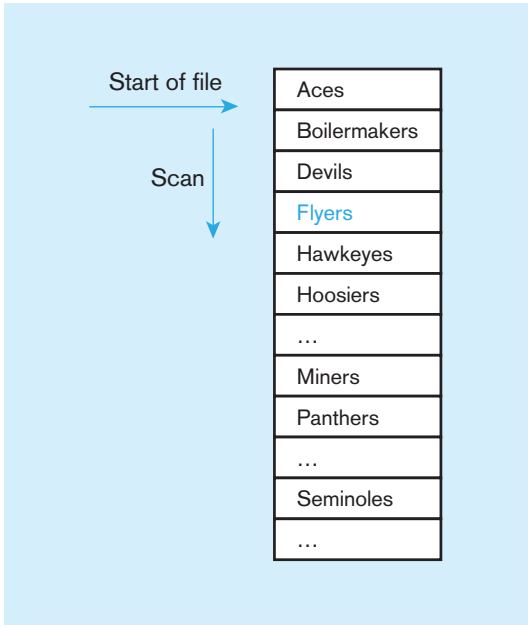
Secondary key
One field or a combination of fields for which more than one record may have the same combination of values. Also called a nonunique key.

INDEXED FILE ORGANIZATIONS In an **indexed file organization**, the records are stored either sequentially or nonsequentially, and an index is created that allows the application software to locate individual records (see Figure 5-7b). Like a card catalog in a library, an **index** is a table that is used to determine in a file the location of records that satisfy some condition. Each index entry matches a key value with one or more records. An index can point to unique records (a primary key index, such as on the ProductID field of a product record) or to potentially more than one record. An index that allows each entry to point to more than one record is called a **secondary key** index. Secondary key indexes are important for supporting many reporting requirements and for providing rapid ad hoc data retrieval. An example would be an index on the ProductFinish column of a Product table. Because indexes are extensively used with relational DBMSs, and the choice of what index and how to store the index entries matters greatly in database processing performance, we review indexed file organizations in more detail than the other types of file organizations.

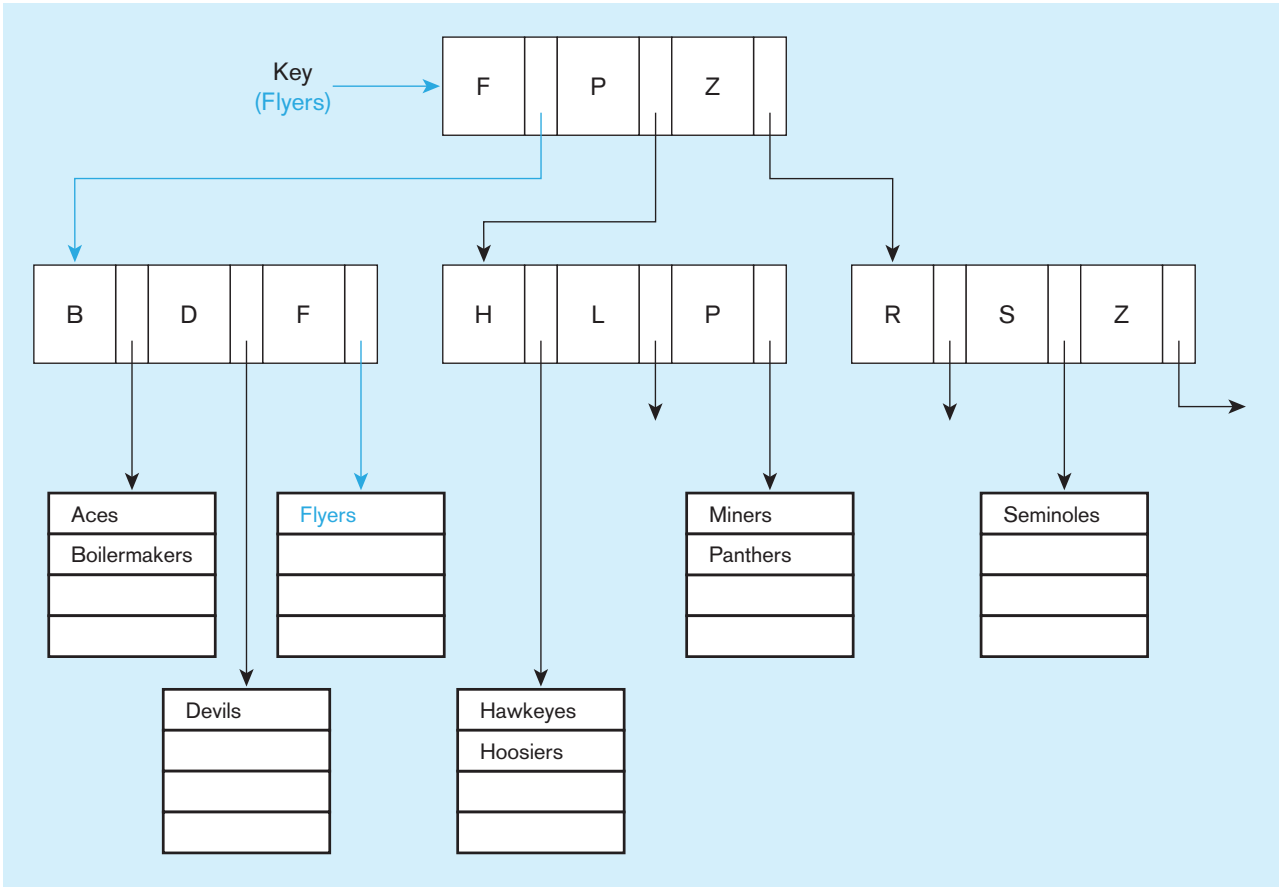
Some index structures influence where table rows are stored, and other index structures are independent of where rows are located. Because the actual structure of an

FIGURE 5-7 Comparison of file organizations

(a) Sequential

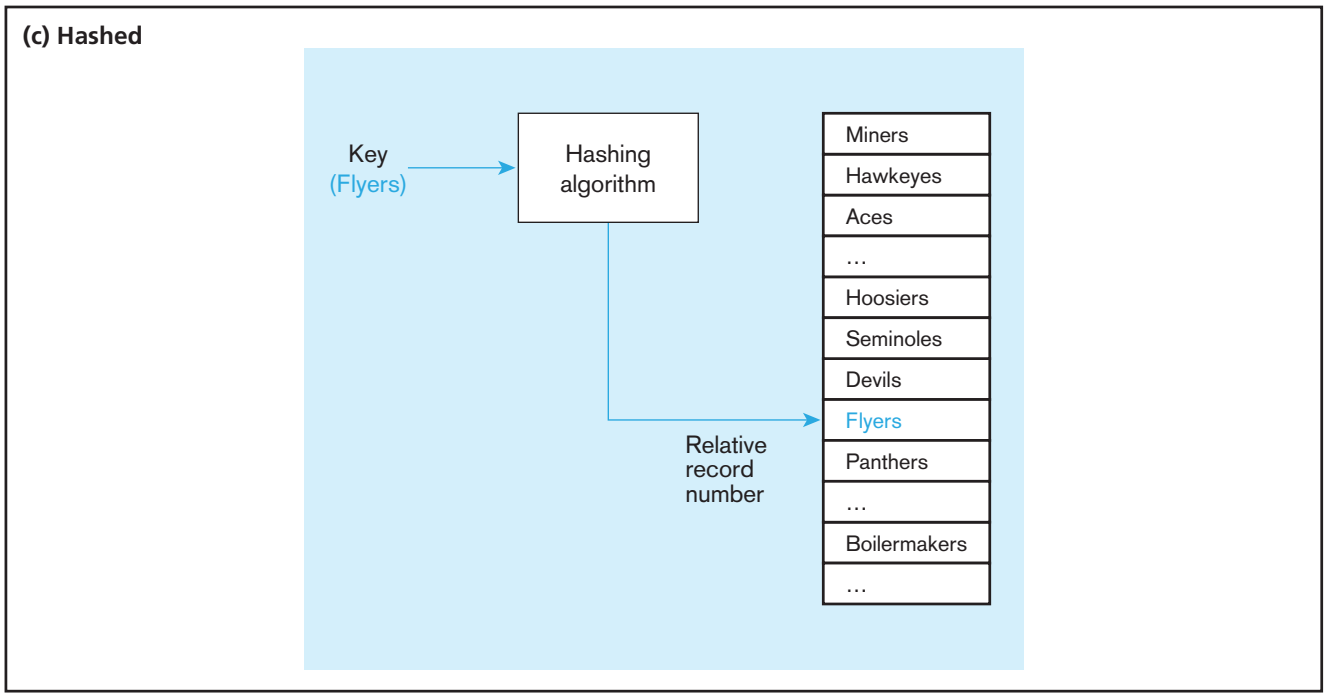


(b) Indexed



(continued)

FIGURE 5-7 ***(continued)***



index does not influence database design and is not important in writing database queries, we will not address the actual physical structure of indexes in this chapter. Thus, Figure 5-7b should be considered a logical view of how an index is used, not a physical view of how data are stored in an index structure.

Transaction processing applications require rapid response to queries that involve one or a few related table rows. For example, to enter a new customer order, an order entry application needs to find the specific customer table row rapidly, a few product table rows for the items being purchased, possibly a few other product table rows based on the characteristics of the products the customer wants (e.g., product finish), and then the application needs to add one customer order and one customer shipment row to the respective tables. The types of indexes discussed so far work very well in an application that is searching for a few specific table rows.

Another increasingly popular type of index, especially in data warehousing and other decision support applications (see Chapter 9), is a join index. In decision support applications, the data accessing tends to want all rows from very large tables that are related to one another (e.g., all the customers who have bought items from the same store). A **join index** is an index on columns from two or more tables that come from the same domain of values. For example, consider Figure 5-8a, which shows two tables, Customer and Store. Each of these tables has a column called City. The join index of the City column indicates the row identifiers for rows in the two tables that have the same City value. Because of the way many data warehouses are designed, there is a high frequency for queries to find data (facts) in common to a store and a customer in the same city (or similar intersections of facts across multiple dimensions). Figure 5-8b shows another possible application for a join index. In this case, the join index precomputes the matching of a foreign key in the Order table with the associated customer in the Customer table (i.e., the result of a relational join operator, which will be discussed in Chapter 6). Simply stated, a join says find rows in the same or different tables that have values that match some criterion.

Join index
An index on columns from two or more tables that come from the same domain of values.

A join index is created as rows are loaded into a database, so the index, like all other indexes previously discussed, is always up-to-date. Without a join index in the database of Figure 5-8a, any query that wants to find stores and customers in the same city would have to compute the equivalent of the join index each time the query is run. For very large tables, joining all the rows of one table with matching rows in another possibly large table can be very time-consuming and can significantly delay responding to an online query. In Figure 5-8b, the join index provides one place for the DBMS to find

Customer

RowID	Cust#	CustName	City	State
10001	C2027	Hadley	Dayton	Ohio
10002	C1026	Baines	Columbus	Ohio
10003	C0042	Ruskin	Columbus	Ohio
10004	C3861	Davies	Toledo	Ohio
. . .				

Store

RowID	Store#	City	Size	Manager
20001	S4266	Dayton	K2	E2166
20002	S2654	Columbus	K3	E0245
20003	S3789	Dayton	K4	E3330
20004	S1941	Toledo	K1	E0874
. . .				

Join Index

CustRowID	StoreRowID	Common Value*
10001	20001	Dayton
10001	20003	Dayton
10002	20002	Columbus
10003	20002	Columbus
10004	20004	Toledo
. . .		

*This column may or may not be included, as needed. Join index could be sorted on any of the three columns. Sometimes two join indexes are created, one as above and one with the two RowID columns reversed.

FIGURE 5-8 Join indexes
(a) Join index for common nonkey columns

(continued)

information about related table rows. A join index, similarly to any other index, saves query processing time by finding data meeting a prespecified qualification at the expense of the extra storage space and maintenance of the index. The use of databases for new applications, such as in data warehousing and online decision support, is leading to the development of new types of indexes. We encourage you to investigate the indexing capabilities of the database management system you are using to understand fully when to apply each type of index and how to tune the performance of the index structures.

HASHED FILE ORGANIZATIONS In a **hashed file organization**, the address of each record is determined using a hashing algorithm (see Figure 5-7c). A **hashing algorithm** is a routine that converts a primary key value into a record address. Although there are several variations of hashed files, in most cases the records are located nonsequentially, as dictated by the hashing algorithm. Thus, sequential data processing is impractical.

Hashed file organization
A storage system in which the address for each record is determined using a hashing algorithm.

Hashing algorithm
A routine that converts a primary key value into a relative record number or relative file address.

A typical hashing algorithm uses the technique of dividing each primary key value by a suitable prime number and then using the remainder of the division as the relative storage location. For example, suppose that an organization has a set of

FIGURE 5-8 (*continued*)
(b) Join index for matching a foreign key (FK) and a primary key (PK)

Order

RowID	Order#	Order Date	Cust#(FK)
30001	O5532	10/01/2001	C3861
30002	O3478	10/01/2001	C1062
30003	O8734	10/02/2001	C1062
30004	O9845	10/02/2001	C2027
. . .			

Customer

RowID	Cust#(PK)	CustName	City	State
10001	C2027	Hadley	Dayton	Ohio
10002	C1062	Baines	Columbus	Ohio
10003	C0042	Ruskin	Columbus	Ohio
10004	C3861	Davies	Toledo	Ohio
. . .				

Join Index

CustRowID	OrderRowID	Cust#
10001	30004	C2027
10002	30002	C1062
10002	30003	C1062
10004	30001	C3861
. . .		

approximately 1,000 employee records to be stored on magnetic disk. A suitable prime number would be 997, because it is close to 1,000. Now consider the record for employee 12,396. When we divide this number by 997, the remainder is 432. Thus, this record is stored at location 432 in the file. Another technique (not discussed here) must be used to resolve duplicates (or overflow) that can occur with the division/remainder method when two or more keys hash to the same address (known as a "hash clash").

One of the severe limitations of hashing is that because data table row locations are dictated by the hashing algorithm, only one key can be used for hashing-based (storage and) retrieval. Hashing and indexing can be combined into what is called a hash index table to overcome this limitation. A **hash index table** uses hashing to map a key into a location in an index (sometimes called a *scatter index table*), where there is a **pointer** (a field of data indicating a target address that can be used to locate a related field or record of data) to the actual data record matching the hash key. The index is the target of the hashing algorithm, but the actual data are stored separately from the addresses generated by hashing. Because the hashing results in a position in an index, the table rows can be stored independently of the hash address, using whatever file organization for the data table makes sense (e.g., sequential or first available space). Thus, as with other indexing schemes but unlike most pure hashing schemes, there can be several primary and secondary keys, each with its own hashing algorithm and index table, sharing one data table.

Hash index table
A file organization that uses hashing to map a key into a location in an index, where there is a pointer to the actual data record matching the hash key.

Pointer
A field of data indicating a target address that can be used to locate a related field or record of data.

Also, because an index table is much smaller than a data table, the index can be more easily designed to reduce the likelihood of key collisions, or overflows, than can

occur in the more space-consuming data table. Again, the extra storage space for the index adds flexibility and speed for data retrieval, along with the added expense of storing and maintaining the index space. Another use of a hash index table is found in some data warehousing database technologies that use parallel processing. In this situation, the DBMS can evenly distribute data table rows across all storage devices to fairly distribute work across the parallel processors, while using hashing and indexing to rapidly find on which processor desired data are stored.

As stated earlier, the DBMS will handle the management of any hashing file organization. You do not have to be concerned with handling overflows, accessing indexes, or the hashing algorithm. What is important for you, as a database designer, is to understand the properties of different file organizations so that you can choose the most appropriate one for the type of database processing required in the database and application you are designing. Also, understanding the properties of the file organizations used by the DBMS can help a query designer write a query in a way that takes advantage of the file organization's properties. As you will see in Chapters 6 and 7, many queries can be written in multiple ways in SQL; different query structures, however, can result in vastly different steps by the DBMS to answer the query. If you know how the DBMS thinks about using a file organization (e.g., what indexes it uses when and how and when it uses a hashing algorithm), you can design better databases and more efficient queries.

The three families of file organizations cover most of the file organizations you will have at your disposal as you design physical files and databases. Although more complex structures can be built using the data structures outlined in Appendix C, you are unlikely to be able to use these with a database management system.

Table 5-3 summarizes the comparative features of sequential, indexed, and hashed file organizations. You should review this table and study Figure 5-7 to see why each comparative feature is true.

Clustering Files

Some database management systems allow adjacent secondary memory space to contain rows from several tables. For example, in Oracle, rows from one, two, or more related tables that are often joined together can be stored so that they share the same data

TABLE 5-3 Comparative Features of Different File Organizations

	File Organization		
Factor	**Sequential**	**Indexed**	**Hashed**
Storage space	No wasted space	No wasted space for data but extra space for index	Extra space may be needed to allow for addition and deletion of records after the initial set of records is loaded
Sequential retrieval on primary key	Very fast	Moderately fast	Impractical, unless using a hash index
Random retrieval on primary key	Impractical	Moderately fast	Very fast
Multiple-key retrieval	Possible but requires scanning whole file	Very fast with multiple indexes	Not possible unless using a hash index
Deleting records	Can create wasted space or require reorganizing	If space can be dynamically allocated, this is easy but requires maintenance of indexes	Very easy
Adding new records	Requires rewriting a file	If space can be dynamically allocated, this is easy but requires maintenance of indexes	Very easy, but multiple keys with the same address require extra work
Updating records	Usually requires rewriting a file	Easy but requires maintenance of indexes	Very easy

blocks (the smallest storage units). A cluster is defined by the tables and the column or columns by which the tables are usually joined. For example, a Customer table and a customer Order table would be joined by the common value of CustomerID, or the rows of a PriceQuote table (which contains prices on items purchased from vendors) might be clustered with the Item table by common values of ItemID. Clustering reduces the time to access related records compared to the normal allocation of different files to different areas of a disk. Time is reduced because related records will be closer to each other than if the records are stored in separate files in separate areas of the disk. Defining a table to be in only one cluster reduces retrieval time for only those tables stored in the same cluster.

The following Oracle database definition commands show how a cluster is defined and tables are assigned to the cluster. First, the cluster (adjacent disk space) is specified, as in the following example:

```
CREATE CLUSTER Ordering (CustomerID CHAR(25));
```

The term Ordering names the cluster space; the attribute CustomerID specifies the attribute with common values.

Then tables are assigned to the cluster when the tables are created, as in the following example:

```
CREATE TABLE Customer_T (
  CustomerID                VARCHAR2(25) NOT NULL,
  CustomerAddress           VARCHAR2(15)
  )
  CLUSTER Ordering (CustomerID);
CREATE TABLE Order_T (
  OrderID                   VARCHAR2(20) NOT NULL,
  CustomerID                VARCHAR2(25) NOT NULL,
  OrderDate                 DATE
  )
  CLUSTER Ordering (CustomerID);
```

Access to records in a cluster can be specified in Oracle to be via an index on the cluster key or via a hashing function on the cluster key. Reasons for choosing an indexed versus a hashed cluster are similar to those for choosing between indexed and hashed files (see Table 5-3). Clustering records is best used when the records are fairly static. When records are frequently added, deleted, and changed, wasted space can arise, and it may be difficult to locate related records close to one another after the initial loading of records, which defines the clusters. Clustering is, however, one option a file designer has to improve the performance of tables that are frequently used together in the same queries and reports.

Designing Controls for Files

One additional aspect of a database file about which you may have design options is the types of controls you can use to protect the file from destruction or contamination or to reconstruct the file if it is damaged. Because a database file is stored in a proprietary format by the DBMS, there is a basic level of access control. You may require additional security controls on fields, files, or databases. We address these options in detail in Chapter 11. Briefly, files will be damaged, so the key is the ability to rapidly restore a damaged file. Backup procedures provide a copy of a file and of the transactions that have changed the file. When a file is damaged, the file copy or current file, along with the log of transactions, is used to recover the file to an uncontaminated state. In terms of security, the most effective method is to encrypt the contents of the file so that only programs with access to the decryption routine will be able to see the file contents. Again, these important topics will be covered later, when you study the activities of data and database administration in Chapter 11.

USING AND SELECTING INDEXES

Most database manipulations require locating a row (or collection of rows) that satisfies some condition. Given the terabyte size of modern databases, locating data without some help would be like looking for the proverbial "needle in a haystack"; or, in more contemporary terms, it would be like searching the Internet without a powerful search engine. For example, we might want to retrieve all customers in a given zip code or all students with a particular major. Scanning every row in a table, looking for the desired rows, may be unacceptably slow, particularly when tables are large, as they often are in real-world applications. Using indexes, as described earlier, can greatly speed up this process, and defining indexes is an important part of physical database design.

As described in the section on indexes, indexes on a file can be created for either a primary or a secondary key or both. It is typical that an index would be created for the primary key of each table. The index is itself a table with two columns: the key and the address of the record or records that contain that key value. For a primary key, there will be only one entry in the index for each key value.

Creating a Unique Key Index

The Customer table defined in the section on clustering has the primary key CustomerID. A unique key index would be created on this field using the following SQL command:

```
CREATE UNIQUE INDEX CustIndex_PK ON Customer_T(CustomerID);
```

In this command, CustIndex_PK is the name of the index file created to store the index entries. The ON clause specifies which table is being indexed and the column (or columns) that forms the index key. When this command is executed, any existing records in the Customer table would be indexed. If there are duplicate values of CustomerID, the CREATE INDEX command will fail. Once the index is created, the DBMS will reject any insertion or update of data in the CUSTOMER table that would violate the uniqueness constraint on CustomerIDs. Notice that every unique index creates overhead for the DBMS to validate uniqueness for each insertion or update of a table row on which there are unique indexes. We will return to this point later, when we review when to create an index.

When a composite unique key exists, you simply list all the elements of the unique key in the ON clause. For example, a table of line items on a customer order might have a composite unique key of OrderID and ProductID. The SQL command to create this index for the OrderLine_T table would be as follows:

```
CREATE UNIQUE INDEX LineIndex_PK ON OrderLine_T(OrderID, ProductID);
```

Creating a Secondary (Nonunique) Key Index

Database users often want to retrieve rows of a relation based on values for various attributes other than the primary key. For example, in a Product table, users might want to retrieve records that satisfy any combination of the following conditions:

- All table products (Description = "Table")
- All oak furniture (ProductFinish = "Oak")
- All dining room furniture (Room = "DR")
- All furniture priced below $500 (Price < 500)

To speed up such retrievals, we can define an index on each attribute that we use to qualify a retrieval. For example, we could create a nonunique index on the Description field of the Product table with the following SQL command:

```
CREATE INDEX DescIndex_FK ON Product_T(Description);
```

Notice that the term UNIQUE should not be used with secondary (nonunique) key attributes, because each value of the attribute may be repeated. As with unique keys, a secondary key index can be created on a combination of attributes.

When to Use Indexes

During physical database design, you must choose which attributes to use to create indexes. There is a trade-off between improved performance for retrievals through the use of indexes and degraded performance (because of the overhead for extensive index maintenance) for inserting, deleting, and updating the indexed records in a file. Thus, indexes should be used generously for databases intended primarily to support data retrievals, such as for decision support and data warehouse applications. Indexes should be used judiciously for databases that support transaction processing and other applications with heavy updating requirements, because the indexes impose additional overhead.

Following are some rules of thumb for choosing indexes for relational databases:

1. Indexes are most useful on larger tables.
2. Specify a unique index for the primary key of each table.
3. Indexes are most useful for columns that frequently appear in WHERE clauses of SQL commands either to qualify the rows to select (e.g., WHERE ProductFinish = "Oak," for which an index on ProductFinish would speed retrieval) or for linking (joining) tables (e.g., WHERE Product_T.ProductID = OrderLine_T.ProductID, for which a secondary key index on ProductID in the OrderLine_T table and a primary key index on ProductID in the Product_T table would improve retrieval performance). In the latter case, the index is on a foreign key in the OrderLine_T table that is used in joining tables.
4. Use an index for attributes referenced in ORDER BY (sorting) and GROUP BY (categorizing) clauses. You do have to be careful, though, about these clauses. Be sure that the DBMS will, in fact, use indexes on attributes listed in these clauses (e.g., Oracle uses indexes on attributes in ORDER BY clauses but not GROUP BY clauses).
5. Use an index when there is significant variety in the values of an attribute. Oracle suggests that an index is not useful when there are fewer than 30 different values for an attribute, and an index is clearly useful when there are 100 or more different values for an attribute. Similarly, an index will be helpful only if the results of a query that uses that index do not exceed roughly 20 percent of the total number of records in the file (Schumacher, 1997).
6. Before creating an index on a field with long values, consider first creating a compressed version of the values (coding the field with a surrogate key) and then indexing on the coded version (Catterall, 2005). Large indexes, created from long index fields, can be slower to process than small indexes.
7. If the key for the index is going to be used for determining the location where the record will be stored, then the key for this index should be a surrogate key so that the values cause records to be evenly spread across the storage space (Catterall, 2005). Many DBMSs create a sequence number so that each new row added to a table is assigned the next number in sequence; this is usually sufficient for creating a surrogate key.
8. Check your DBMS for the limit, if any, on the number of indexes allowable per table. Some systems permit no more than 16 indexes and may limit the size of an index key value (e.g., no more than 2,000 bytes for each composite value). If there is such a limit in your system, you will have to choose those secondary keys that will most likely lead to improved performance.
9. Be careful of indexing attributes that have null values. For many DBMSs, rows with a null value will not be referenced in the index (so they cannot be found from an *index search* of the attribute = NULL). Such a search will have to be done by scanning the file.

Selecting indexes is arguably the most important physical database design decision, but it is not the only way you can improve the performance of a database. Other ways address such issues as reducing the costs to relocate records, optimizing the use of extra or so-called free space in files, and optimizing query processing algorithms. (See Viehman, 1994, for a discussion of these additional ways to enhance physical database design and efficiency.) We briefly discuss the topic of query optimization in the following section of

this chapter because such optimization can be used to overrule how the DBMS would use certain database design options included because of their expected improvement in data processing performance in most instances.

DESIGNING A DATABASE FOR OPTIMAL QUERY PERFORMANCE

The primary purpose today for physical database design is to optimize the performance of database processing. Database processing includes adding, deleting, and modifying a database, as well as a variety of data retrieval activities. For databases that have greater retrieval traffic than maintenance traffic, optimizing the database for query performance (producing online or off-line anticipated and ad hoc screens and reports for end users) is the primary goal. This chapter has already covered most of the decisions you can make to tune the database design to meet the need of database queries (clustering, indexes, file organizations, etc.). In this final section of this chapter, we introduce parallel query processing as an additional advanced database design and processing option now available in many DBMSs.

The amount of work a database designer needs to put into optimizing query performance depends greatly on the DBMS. Because of the high cost of expert database developers, the less database and query design work developers have to do, the less costly the development and use of a database will be. Some DBMSs give very little control to the database designer or query writer over how a query is processed or the physical location of data for optimizing data reads and writes. Other systems give the application developers considerable control and often demand extensive work to tune the database design and the structure of queries to obtain acceptable performance. Sometimes, the workload varies so much and the design options are so subtle that good performance is all that can be achieved. When the workload is fairly focused—say, for data warehousing, where there are a few batch updates and very complex queries requiring large segments of the database—performance can be well tuned either by smart query optimizers in the DBMS or by intelligent database and query design or a combination of both. For example, the Teradata DBMS is highly tuned for parallel processing in a data warehousing environment. In this case, rarely can a database designer or query writer improve on the capabilities of the DBMS to store and process data. This situation is, however, rare, and therefore it is important for a database designer to consider options for improving database processing performance. Chapter 7 will provide additional guidelines for writing efficient queries.

Parallel Query Processing

One of the major computer architectural changes over the past few years is the increased use of multiple processors in database servers. Database servers frequently use symmetric multiprocessor (SMP) technology (Schumacher, 1997). To take advantage of this parallel processing capability, some of the most sophisticated DBMSs include strategies for breaking apart a query into modules that can be processed in parallel by each of the related processors. The most common approach is to replicate the query so that each copy works against a portion of the database, usually a horizontal partition (i.e., sets of rows). The partitions need to be defined in advance by the database designer. The same query is run against each portion in parallel on separate processors, and the intermediate results from each processor are combined to create the final query result as if the query were run against the whole database.

Suppose you have an Order table with several million rows for which query performance has been slow. To ensure that subsequent scans of this table are performed in parallel, using at least three processors, you would alter the structure of the table with the SQL command:

```
ALTER TABLE Order_T PARALLEL 3;
```

You need to tune each table to the best degree of parallelism, so it is not uncommon to alter a table several times until the right degree is found.

Parallel query processing speed can be impressive. Schumacher (1997) reports on a test in which the time to perform a query was cut in half with parallel processing

compared to using a normal table scan. Because an index is a table, indexes can also be given the parallel structure, so that scans of an index are also faster. Again, Schumacher (1997) shows an example where the time to create an index by parallel processing was reduced from approximately seven minutes to five seconds!

Besides table scans, other elements of a query can be processed in parallel, such as certain types of joining related tables, grouping query results into categories, combining several parts of a query result together (called *union*), sorting rows, and computing aggregate values. Row update, delete, and insert operations can also be processed in parallel. In addition, the performance of some database creation commands can be improved by parallel processing; these include creating and rebuilding an index and creating a table from data in the database. The Oracle environment must be preconfigured with a specification for the number of virtual parallel database servers to exist. Once this is done, the query processor will decide what it thinks is the best use of parallel processing for any command.

Sometimes the parallel processing is transparent to the database designer or query writer. With some DBMSs, the part of the DBMS that determines how to process a query, the query optimizer, uses physical database specifications and characteristics of the data (e.g., a count of the number of different values for a qualified attribute) to determine whether to take advantage of parallel processing capabilities.

Overriding Automatic Query Optimization

Sometimes, the query writer knows (or can learn) key information about the query that may be overlooked or unknown to the query optimizer module of the DBMS. With such key information in hand, a query writer may have an idea for a better way to process a query. But before you as the query writer can know you have a better way, you have to know how the query optimizer (which usually picks a query processing plan that will minimize expected query processing time, or cost) will process the query. This is especially true for a query you have not submitted before. Fortunately, with most relational DBMSs, you can learn the optimizer's plan for processing the query before running the query. A command such as EXPLAIN or EXPLAIN PLAN (the exact command varies by DBMS) will display how the query optimizer intends to access indexes, use parallel servers, and join tables to prepare the query result. If you preface the actual relational command with the explain clause, the query processor displays the logical steps to process the query and stops processing before actually accessing the database. The query optimizer chooses the best plan based on statistics about each table, such as average row length and number of rows. It may be necessary to force the DBMS to calculate up-to-date statistics about the database (e.g., the Analyze command in Oracle) to get an accurate estimate of query costs. You may submit several EXPLAIN commands with your query, written in different ways, to see if the optimizer predicts different performance. Then, you can submit for actual processing the form of the query that had the best predicted processing time, or you may decide not to submit the query because it will be too costly to run.

You may even see a way to improve query processing performance. With some DBMSs, you can force the DBMS to do the steps differently or to use the capabilities of the DBMS, such as parallel servers, differently than the optimizer thinks is the best plan.

For example, suppose we wanted to count the number of orders processed by a particular sales representative, Smith. In Oracle, parallel table processing works only when a table is scanned, not when it is accessed via an index. So, in Oracle, we might want to force both a full table scan as well as scanning in parallel. The SQL command for this query would be as follows:

```
SELECT /*+ FULL(Order_T) PARALLEL(Order_T,3) */ COUNT(*)
FROM Order_T
WHERE Salesperson = "SMITH";
```

The clause inside the /* */ delimiters is the hint to Oracle. This hint overrides whatever query plan Oracle would naturally create for this query. Thus, a hint is specific to each query, but the use of such hints must be anticipated by altering the structure of tables to be handled with parallel processing.

Summary

During physical database design, you, the designer, translate the logical description of data into the technical specifications for storing and retrieving data. The goal is to create a design for storing data that will provide adequate performance and ensure database integrity, security, and recoverability. In physical database design, you consider normalized relations and data volume estimates, data definitions, data processing requirements and their frequencies, user expectations, and database technology characteristics to establish the specifications that are used to implement the database using a database management system.

A field is the smallest unit of application data, corresponding to an attribute in the logical data model. You must determine the data type, integrity controls, and how to handle missing values for each field, among other factors. A data type is a detailed coding scheme for representing organizational data. Data may be coded to reduce storage space. Field integrity control includes specifying a default value, a range of permissible values, null value permission, and referential integrity.

A process of denormalization transforms normalized relations into non-normalized implementation specifications. Denormalization is done to improve the efficiency of I/O operations by specifying the database implementation structure so that data elements that are required together are also accessed together on the physical medium. Partitioning is also considered a form of denormalization. Horizontal partitioning breaks a relation into multiple record specifications by placing different rows into different tables, based on common column values. Vertical partitioning distributes the columns of a relation into separate files, repeating the primary key in each of the files.

A physical file is a named portion of secondary memory allocated for the purpose of storing physical records. Data within a physical file are organized through a combination of sequential storage and pointers. A pointer is a field of data that can be used to locate a related field or record of data.

A file organization arranges the records of a file on a secondary storage device. The three major categories of file organizations are (1) sequential, which stores records in sequence according to a primary key value; (2) indexed, in which records are stored sequentially or nonsequentially and an index is used to keep track of where the records are stored; and (3) hashed, in which the address of each record is determined using an algorithm that converts a primary key value into a record address. Physical records of several types can be clustered together into one physical file in order to place records frequently used together close to one another in secondary memory.

The indexed file organization is one of the most popular in use today. An index may be based on a unique key or a secondary (nonunique) key, which allows more than one record to be associated with the same key value. A join index indicates rows from two or more tables that have common values for related fields. A hash index table makes the placement of data independent of the hashing algorithm and permits the same data to be accessed via several hashing functions on different fields. Indexes are important in speeding up data retrieval, especially when multiple conditions are used for selecting, sorting, or relating data. Indexes are useful in a wide variety of situations, including for large tables, for columns that are frequently used to qualify the data to be retrieved, when a field has a large number of distinct values, and when data processing is dominated by data retrieval rather than data maintenance.

The introduction of multiprocessor database servers has made possible new capabilities in database management systems. One major new feature is the ability to break apart a query and process the query in parallel against segments of a table. Such parallel query processing can greatly improve the speed of query processing. Also, database programmers can improve database processing performance by providing the DBMS with hints about the sequence in which to perform table operations. These hints override the cost-based optimizer of the DBMS. Both the DBMS and programmers can look at statistics about the database to determine how to process a query. A wide variety of guidelines for good query design were included in the chapter.

This chapter concludes the database design section of this book. Having developed complete physical data specifications, you are now ready to begin implementing the database with database technology. Implementation means defining the database and programming client and server routines to handle queries, reports, and transactions against the database. These are primary topics of the next five chapters, which cover relational database implementation on client platforms, server platforms, client/server environments, and data warehouse technologies.

Chapter Review

Key Terms

Data type *212*
Denormalization *215*
Extent *221*
Field *211*
File organization *221*
Hash index table *226*
Hashed file organization *225*
Hashing algorithm *225*
Horizontal partitioning *218*
Index *222*
Indexed file organization *222*
Join index *224*
Physical file *220*
Pointer *226*
Secondary key *222*
Sequential file organization *222*
Tablespace *221*
Vertical partitioning *220*

Review Questions

1. Define each of the following terms:
   a. file organization
   b. sequential file organization
   c. indexed file organization
   d. hashing file organization
   e. denormalization
   f. composite key
   g. secondary key
   h. data type
   i. join index
2. Match the following terms to the appropriate definitions:

____ extent	a. a detailed coding scheme for representing organizational data
____ hashing algorithm	b. a data structure used to determine in a file the location of a record/records
____ index	c. a named area of secondary memory
____ physical record	d. a contiguous section of disk storage space
____ pointer	e. a field not containing business data
____ data type	f. converts a key value into an address
____ physical file	g. adjacent fields

3. Contrast the following terms:
   a. horizontal partitioning; vertical partitioning
   b. physical file; tablespace
   c. normalization; denormalization
   d. range control; null control
   e. secondary key; primary key
4. What are the major inputs into physical database design?
5. What are the key decisions in physical database design?
6. What decisions have to be made to develop a field specification?
7. Explain how physical database design has an important role in forming a foundation for regulatory compliance.
8. What are the objectives of selecting a data type for a field?
9. Explain why you sometimes have to reserve much more space for a numeric field than any of the initial stored values requires.
10. Why are field values sometimes coded?
11. What options are available for controlling data integrity at the field level?
12. Describe three ways to handle missing field values.
13. Explain why normalized relations may not comprise an efficient physical implementation structure.
14. List three common situations that suggest that relations be denormalized before database implementation.
15. Explain the reasons why some observers are against the practice of denormalization.
16. What are the advantages and disadvantages of horizontal and vertical partitioning?
17. List seven important criteria in selecting a file organization.
18. What are the benefits of a hash index table?
19. What is the purpose of clustering of data in a file?
20. State nine rules of thumb for choosing indexes.
21. Indexing can clearly be very beneficial. Why should you *not* create an index for every column of every table of your database?
22. Explain how parallel processing can improve query performance.

Problems and Exercises

1. Consider the following two relations for Millennium College:

```
STUDENT(StudentID, StudentName,
   CampusAddress, GPA)
REGISTRATION(StudentID, CourseID, Grade)
```

Following is a typical query against these relations:

```
SELECT Student_T.StudentID, StudentName,
   CourseID, Grade
FROM Student_T, Registration_T
      WHERE Student_T.StudentID =
        Registration_T.StudentID
      AND GPA > 3.0
ORDER BY StudentName;
```

   a. On what attributes should indexes be defined to speed up this query? Give the reasons for each attribute selected.
   b. Write SQL commands to create indexes for each attribute you identified in part a.

Problems and Exercises 2–5 have been written assuming that the DBMS you are using is Oracle. If that is not the case, feel free to modify the question for the DBMS environment that you are familiar with. You can also compare and contrast answers for different DBMSs.

2. Choose Oracle data types for the attributes in the normalized relations in Figure 5-4b.
3. Choose Oracle data types for the attributes in the normalized relations that you created in Problem and Exercise 19 in Chapter 4.
4. Explain in your own words what the precision (p) and scale (s) parameters for the Oracle data type NUMBER mean.
5. Say that you are interested in storing the numeric value 3,456,349.2334. What will be stored, with each of the following Oracle data types:
   a. NUMBER(11)
   b. NUMBER(11,1)
   c. NUMBER(11,-2)
   d. NUMBER(6)
   e. NUMBER
6. Suppose you are designing a default value for the age field in a student record at your university. What possible values would you consider, and why? How might the default vary by other characteristics about the student, such as school within the university or degree sought?
7. When a student has not chosen a major at a university, the university often enters a value of "Undecided" for the major field. Is "Undecided" a way to represent the null value? Should it be used as a default value? Justify your answer carefully.

8. Consider the following normalized relations from a database in a large retail chain:

```
STORE (StoreID, Region, ManagerID, SquareFeet)
EMPLOYEE (EmployeeID, WhereWork, EmployeeName,
   EmployeeAddress)
DEPARTMENT (DepartmentID, ManagerID, SalesGoal)
SCHEDULE (DepartmentID, EmployeeID, Date)
```

   What opportunities might exist for denormalizing these relations when defining the physical records for this database? Under what circumstances would you consider creating such denormalized records?

9. Consider the following normalized relations for a sports league:

```
TEAM(TeamID, TeamName, TeamLocation)
PLAYER(PlayerID, PlayerFirstName, PlayerLastName,
   PlayerDateOfBirth, PlayerSpecialtyCode)
SPECIALTY(SpecialtyCode, SpecialtyDescription)
CONTRACT(TeamID, PlayerID, StartTime, EndTime, Salary)
LOCATION(LocationID, CityName, CityState,
   CityCountry, CityPopulation)
MANAGER(ManagerID, ManagerName, ManagerTeam)
```

   What recommendations would you make regarding opportunities for denormalization? What additional information would you need to make fully informed denormalization decisions?

10. What problems might arise from vertically partitioning a relation? Given these potential problems, what general conditions influence when to partition a relation vertically?

11. Is it possible with a sequential file organization to permit sequential scanning of the data, based on several sorted orders? If not, why not? If it is possible, how?

12. Suppose each record in a file were connected to the prior record and the next record in key sequence using pointers. Thus, each record might have the following format:

    Primary key, other attributes, pointer to prior record, pointer to next record

    a. What would be the advantages of this file organization compared with a sequential file organization?
    b. In contrast with a sequential file organization, would it be possible to keep the records in multiple sequences? Why or why not?

13. Assume that a student table in a university database had an index on StudentID (the primary key) and indexes on Major, Age, MaritalStatus, and HomeZipCode (all secondary keys). Further, assume that the university wanted a list of students majoring in MIS or computer science, over age 25, and married OR students majoring in computer engineering, single, and from the 45462 zip code. How could indexes be used so that only records that satisfy this qualification are accessed?

14. Consider Figure 5-7b. Assuming that the empty rows in the leaves of this index show space where new records can be stored, explain where the record for Sooners would be stored. Where would the record for Flashes be stored? What might happen when one of the leaves is full and a new record needs to be added to that leaf?

15. Consider Figure 4-36 and your answer to Problem and Exercise 19 in Chapter 4. Assume that the most important reports that the organization needs are as follows:
    - A list of the current developer's project assignments
    - A list of the total costs for all projects
    - For each team, a list of its membership history
    - For each country, a list of all projects, with projected end dates, in which the country's developers are involved
    - For each year separately, a list of all developers, in the order of their average assignment scores for all the assignments that were completed during that year

    Based on this (admittedly limited) information, make a recommendation regarding the indexes that you would create for this database. Choose two of the indexes and provide the SQL command that you would use to create those indexes.

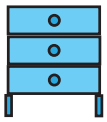
16. Can clustering of files occur after the files are populated with records? Why or why not?

17. Parallel query processing, as described in this chapter, means that the same query is run on multiple processors and that each processor accesses in parallel a different subset of the database. Another form of parallel query processing, not discussed in this chapter, would partition the query so that each part of the query runs on a different processor, but that part accesses whatever part of the database it needs. Most queries involve a qualification clause that selects the records of interest in the query. In general, this qualification clause is of the following form:

    (condition OR condition OR . . .) AND (condition OR condition OR . . .) AND . . .

    Given this general form, how might a query be broken apart so that each parallel processor handles a subset of the query and then combines the subsets together after each part is processed?

Problems and Exercises 18–21 refer to the large Pine Valley Furniture Company data set provided with the text.



PINE VALLEY FURNITURE

18. Create a join index on the CustomerID fields of the Customer_T and Order_T tables in Figure 4-4.

19. Consider the composite usage map in Figure 5-1. After a period of time, the assumptions for this usage map have changed, as follows:
    - There is an average of 40 supplies (rather than 50) for each supplier.
    - Manufactured parts represent only 30 percent of all parts, and purchased parts represent 75 percent.
    - The number of direct access to purchased parts increases to 7,500 per hour (rather than 6,000).

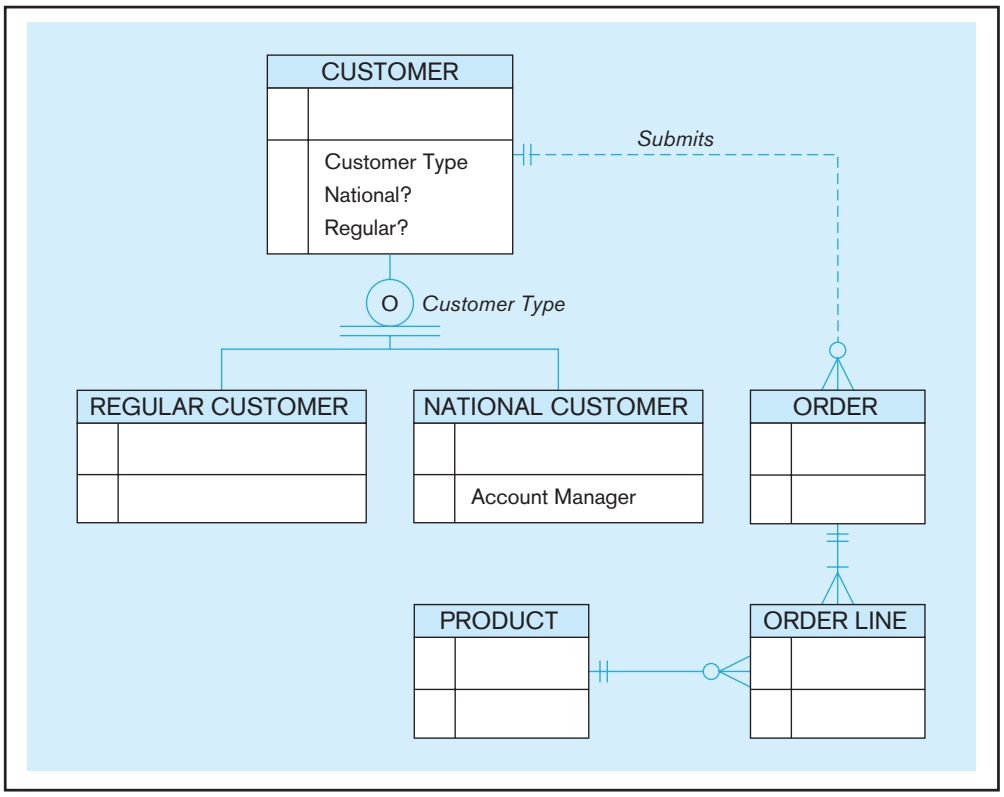
    Draw a new composite usage map reflecting this new information to replace Figure 5-1.

20. Consider the EER diagram for Pine Valley Furniture shown in Figure 3-12. Figure 5-9 looks at a portion of that EER diagram.

    Let's make a few assumptions about the average usage of the system:
    - There are 50,000 customers, and of these, 80 percent represent regular accounts and 20 percent represent national accounts.
    - Currently, the system stores 800,000 orders, although this number is constantly changing.
    - Each order has an average of 20 products.
    - There are 3,000 products.
    - Approximately 500 orders are placed per hour.

    a. Based on these assumptions, draw a usage map for this portion of the EER diagram.
    b. Management would like employees only to use this database. Do you see any opportunities for denormalization?

FIGURE 5-9 Figure for Problem and Exercise 20



21. Refer to Figure 4-5. For each of the following reports (with sample data), indicate any indexes that you feel would help the report run faster as well as the type of index:

a. State, by products (user-specified period)

State, by Products Report, January 1, 2010, to March 31, 2010

State	Product Description	Total Quantity Ordered
CO	8-Drawer Dresser	1
CO	Entertainment Center	0
CO	Oak Computer Desk	1
CO	Writer's Desk	2
NY	Writer's Desk	1
VA	Writer's Desk	5

b. Most frequently sold product finish in a user-specified month

Most Frequently Sold Product Finish Report, March 1, 2010, to March 31, 2010

Product Finish	Units Sold
Cherry	13

c. All orders placed last month

Monthly Order Report, March 1, 2010, to March 31, 2010

Order ID	Order Date	Customer ID	Customer Name
19	3/5/10	4	Eastern Furniture

Associated Order Details:

Product Description	Quantity Ordered	Price	Extended Price
Cherry End Table	10	$75.00	$750.00
High Back Leather Chair	5	$362.00	$1,810.00

Order_ID	Order Date	Customer IDs	Customer Name
24	3/10/10	1	Contemporary Casuals

Associated Order Details:

Product Description	Quantity Ordered	Price	Extended Price
Bookcase	4	$69.00	$276.00

d. Total products sold, by product line (user-specified period)

Products Sold by Product Line, March 1, 2010, to March 31, 2010

Product Line	Quantity Sold
Basic	200
Antique	15
Modern	10
Classical	75

Field Exercises

1. Find out which database management systems are available at your university for student use. Investigate which data types these DBMSs support. Compare these DBMSs based on the data types supported and suggest which types of applications each DBMS is best suited for, based on this comparison.
2. Using the Web site for this text and other Internet resources, investigate the parallel processing capabilities of several leading DBMSs. How do their capabilities differ?
3. Denormalization can be a controversial topic among database designers. Some believe that any database should be fully normalized (even using all the normal forms discussed in Appendix B). Others look for ways to denormalize to improve processing performance. Contact a database designer or administrator in an organization with which you are familiar. Ask whether he or she believes in fully normalized or denormalized physical databases. Ask the person why he or she has this opinion.
4. Contact a database designer or administrator in an organization with which you are familiar. Ask what file organizations are available in the various DBMSs used in that organization. Interview this person to learn what factors he or she considers when selecting an organization for database files. For indexed files, ask how he or she decides what indexes to create. Are indexes ever deleted? Why or why not?

References

Babad, Y. M., and J. A. Hoffer. 1984. "Even No Data Has a Value." *Communications of the ACM* 27,8 (August): 748–56.

Bieniek, D. 2006. "The Essential Guide to Table Partitioning and Data Lifecycle Management." *Windows IT Pro* (March) accessed at **www.windowsITpro.com**.

Brobst, S., S. Gant, and F. Thompson. 1999. "Partitioning Very Large Database Tables with Oracle8." *Oracle Magazine* 8,2 (March–April): 123–26.

Catterall, R. 2005. "The Keys to the Database." *DB2 Magazine* 10,2 (Quarter 2): 49–51.

Finkelstein, R. 1988. "Breaking the Rules Has a Price." *Database Programming & Design* 1,6 (June): 11–14.

Hoberman, S. 2002. "The Denormalization Survival Guide—Parts I and II." Published in the online journal *The Data Administration Newsletter*, found in the April and July issues of Tdan.com; the two parts of this guide are available at **www.tdan.com/i020fe02.htm** and **www.tdan.com/i021ht03.htm**, respectively.

Inmon, W. H. 1988. "What Price Normalization." *ComputerWorld* (October 17): 27, 31.

Pascal, F. 2002a. "The Dangerous Illusion: Denormalization, Performance and Integrity, Part 1." *DM Review* 12,6 (June): 52–53, 57.

Pascal, F. 2002b. "The Dangerous Illusion: Denormalization, Performance and Integrity, Part 2." *DM Review* 12,6 (June): 16, 18.

Rogers, U. 1989. "Denormalization: Why, What, and How?" *Database Programming & Design* 2,12 (December): 46–53.

Schumacher, R. 1997. "Oracle Performance Strategies." *DBMS* 10,5 (May): 89–93.

Viehman, P. 1994. "Twenty-four Ways to Improve Database Performance." *Database Programming & Design* 7,2 (February): 32–41.

Further Reading

Ballinger, C. 1998. "Introducing the Join Index." *Teradata Review* 1,3 (Fall): 18–23. (Note: *Teradata Review* is now *Teradata Magazine*.)

Bontempo, C. J., and C. M. Saracco. 1996. "Accelerating Indexed Searching." *Database Programming & Design* 9,7 (July): 37–43.

DeLoach, A. 1987. "The Path to Writing Efficient Queries in SQL/DS." *Database Programming & Design* 1,1 (January): 26–32.

Elmasri, R., and S. Navathe. 2006. *Fundamentals of Database Systems*, 5th ed. Menlo Park, CA: Benjamin Cummings.

Loney, K., E. Aronoff, and N. Sonawalla. 1996. "Big Tips for Big Tables." *Database Programming & Design* 9,11 (November): 58–62.

Oracle. 2008. *Oracle SQL Parallel Execution*. An Oracle White Paper, June 2008. Available at **www.oracle.com/technology/products/bi/db/11g/pdf/twp_bidw_parallel_execution_11gr1.pdf**

Roti, S. 1996. "Indexing and Access Mechanisms." *DBMS* 9,5 (May): 65–70.

Web Resources

www.SearchOracle.com and www.SearchSQLServer.com Sites that contain a wide variety of information about database management and DBMSs. New "tips" are added daily, and you can subscribe to an alert service for new postings to the site. Many tips deal with improving the performance of queries through better database and query design.

www.tdan.com Web site of *The Data Administration Newsletter*, which frequently publishes articles on all aspects of database development and design.

www.teradata.com/tdmo/ A journal for NCR Teradata data warehousing products that includes articles on database design. You can search the site for key terms from this chapter, such as *join index*, and find many articles on these topics.

CASE

Mountain View Community Hospital

Case Description

Up to this point, you have developed the conceptual and logical models for Mountain View Community Hospital's database. After considering several options, the hospital has decided to use Microsoft SQL Server, a relational DBMS, for implementing the database. Before the functional database is actually created, it is necessary to specify its physical design to ensure that the database is effective and efficient. As you have learned, physical database design is specific to the target environment and must conform to the capabilities of the DBMS to be used. It requires a good understanding of the DBMS's features, such as available data types, indexing, support for referential integrity and other constraints, and many more. (You can alternatively assume that MVCH chose another DBMS with which you are familiar and then answer the following questions accordingly.)

Case Questions

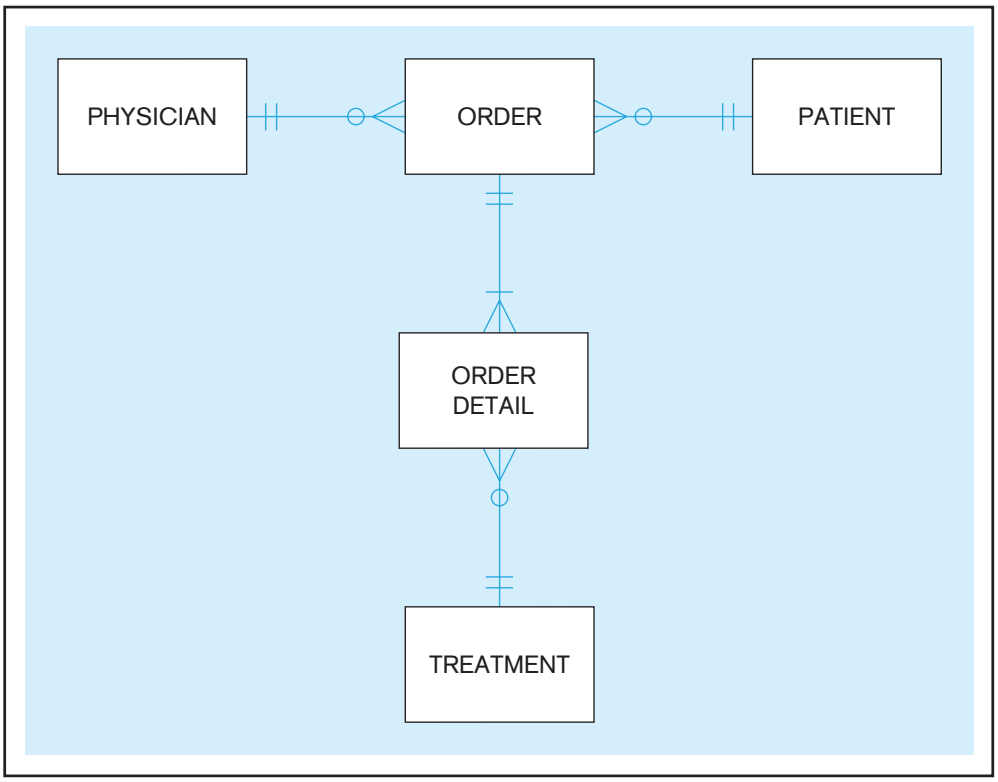
1. What additional kinds of information do you need for the physical database design of the MVCH database besides the 3NF relations you developed earlier for this case in Chapter 4?
2. What different types or forms of clinical data are collected at a hospital such as MVCH? Can you identify data that may not be easily accommodated by the standard data types provided by a DBMS? How would you handle that?
3. Are there opportunities for horizontal or vertical partitioning of this database? If you are not sure, what other information would you need to answer this question with greater certainty?
4. Do you see an opportunity for using a join index for this database? Why or why not?
5. Consider the following query against the MVCH database:

   For each treatment ordered in the past two weeks, list by treatment ID and date (in reverse chronological order) the number of times a physician performed that treatment that day, sorted alphabetically by physician name.

   a. Which secondary key indexes would you suggest to optimize the performance of this query? Why? Make any assumptions you need in order to answer this question.
   b. Following the examples in this chapter, write the SQL statements that create these secondary key indexes.
6. This chapter describes the 2002 Sarbanes-Oxley Act, which is not focused on not-for-profit providers such as many community hospitals.
   a. Can you see how MVCH could benefit from voluntarily complying with SOX?
   b. Specifically how can proper physical database design help with compliance and the following:
      - Improving accuracy and completeness of MVCH data
      - Eliminating duplicates and data inconsistencies
      - Improving understandability of MVCH data

Case Exercises

1. In Case Exercise 2 in Chapter 4, you wrote CREATE TABLE commands for each relation of Dr. Z's small database, which was to be created in Microsoft Access. Since then, Dr. Z has decided to use Microsoft SQL Server, consistent with other databases at MVCH. Reconsider your previous CREATE TABLE commands in answering the following questions:
   a. Would you choose different data types for any fields? Why?
   b. Are any fields candidates for coding? If so, what coding scheme would you use for each of these fields?
   c. Which fields require data values? Are there any fields that may take on null values?
   d. Suppose the *reason for a visit* or *the patient's social worker* are not entered. What procedures would you use for handling these missing data? Can you and should you use a default value for this field? Why or why not?
   e. Using Microsoft Visio (or other tool required by your instructor), draw the physical data model that shows the data types, primary keys, and foreign keys.
2. In Case Exercise 3 from Chapter 4, you developed the relational schema for Dr. Z's Multiple Sclerosis (MS) Clinic Management System.
   a. Do you see any opportunities for user-defined data types? Which fields? Why?
   b. Are any fields candidates for coding? If so, what coding scheme would you use for each of these fields?
   c. Are there any fields that may take on a null value? If so, which ones?
   d. Do you see any opportunities for denormalization of the relations you designed in Chapter 4? If not, why not? If yes, where and how might you denormalize?
   e. Do you see an opportunity for using a bitmap index for this database? Why or why not?
   f. Can you think of a situation with this set of tables where you might want to use a join index?
3. MVCH Figure 5-1 shows a portion of the data model for MVCH's database that represents a set of normalized relations based on the enterprise model shown in MVCH Figure 1-1 and additional business rules provided in the Chapter 2 case segment. Recall that TREATMENT refers to any test or procedure ordered by a physician for a patient and that ORDER refers to any order issued by a physician for treatment and/or services such as diagnostic tests (radiology, laboratory).



MVCH FIGURE 5-1 Partial data model

Using the information provided below regarding data volume and access frequencies, and following the example provided in Figure 5-1, modify the E-R model shown in MVCH Figure 5-1 to create a preliminary composite usage map.

a. Data volume analysis:
- Recall from an earlier case segment that the hospital performs more than a million laboratory procedures and more than 110,000 radiology procedures annually. Add these two figures to arrive at the number of records for the ORDER DETAIL table.
- There are approximately 250 PHYSICIANS, 20,000 PATIENTS, and 200,000 physician ORDERS in this database.
- ICD-9 procedure codes for treatments (lab procedures, radiology procedures, etc.) fall into approximately 3,500 major categories. Use this number to approximate the number of TREATMENT records.

b. Data access frequencies per hour:
- Across all applications that use the MVCH database, there are approximately 100 direct accesses to PHYSICIAN, 35 to ORDER, 200 to PATIENT, and 150 to TREATMENT.
- Of the 200 accesses to PATIENT, 30 accesses then also require ORDER data, and of these 30, there are 20 subsequent accesses to PHYSICIAN, and 30 accesses to ORDER DETAIL.
- Of the 35 direct accesses to ORDER, 10 accesses then also require PHYSICIAN data, and 20 require access to PATIENT data, ORDER DETAIL data, and TREATMENT data.
- Of the 100 direct accesses to PHYSICIAN, 20 also access ORDER, ORDER DETAIL, and TREATMENT data.
- Of the 150 direct accesses to TREATMENT, 10 also access ORDER DETAIL data and associated ORDER and PHYSICIAN data.

4. In Case Exercise 3, you created a composite usage map for part of the MVCH database, based on MVCH Figure 5-1. Referring to that composite usage map, do you see any opportunities for clustering rows from two or more tables? Why or why not? Is the concept of clustering tables supported in SQL Server? Does it differ from Oracle's implementation? If so, how?

PROJECT ASSIGNMENTS

In Chapter 4, you created the relational schema for the MVCH database. Next, you will develop the specification for database implementation. Specifically, you need to identify and document choices regarding the properties of each data element in the database, using the information provided in the case segments and options available in SQL Server (or other DBMS you may be using for this assignment).

P1. Review the information provided in the case segments and identify the data type for each field in the database.
- Do you see any opportunities for user-defined data types? Which fields? Why?
- Are any fields candidates for coding? If so, what coding scheme would you use for each of these fields?
- Which fields may take on a null value? Why?
- Which fields should be indexed? What type of index?

P2. Create a data dictionary similar to the metadata table shown in Table 1-1 in Chapter 1 to document your choices. For each table in the relational schema you developed earlier, provide the following information for each field/data element: field name, definition/description, data type, format, allowable values, whether the field is required or optional, whether the field is indexed and the type of index, whether the field is a primary key, whether the field is a foreign key, and the table that is referenced by the foreign key field.

P3. Using Microsoft Visio (or similar tool designated by your instructor), create the physical data model for the MVCH relational schema you developed in Chapter 4, clearly indicating data types, primary keys, and foreign keys.

P4. Identify five reports to be generated by the database and create a composite usage map for each.