ENTITY-RELATIONSHIP MODEL

The great successful men of the world have used their imaginations. They think ahead and create their mental picture, and then go to work materializing that picture in all its details, filling in here, adding a little there, altering this bit and that bit, but steadily building, steadily building.

—Robert Collier

The entity-relationship (ER) data model allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design. In this chapter, we introduce the ER model and discuss how its features allow us to model a wide range of data faithfully.

The ER model is important primarily for its role in database design. It provides useful concepts that allow us to move from an informal description of what users want from their database to a more detailed, and precise, description that can be implemented in a DBMS. We begin with an overview of database design in Section 2.1 in order to motivate our discussion of the ER model. Within the larger context of the overall design process, the ER model is used in a phase called conceptual database design. We then introduce the ER model in Sections 2.2, 2.3, and 2.4. In Section 2.5, we discuss database design issues involving the ER model. We conclude with a brief discussion of conceptual database design for large enterprises.

We note that many variations of ER diagrams are in use, and no widely accepted standards prevail. The presentation in this chapter is representative of the family of ER models and includes a selection of the most popular features.

2.1 OVERVIEW OF DATABASE DESIGN

The database design process can be divided into six steps. The ER model is most relevant to the first three steps:

1) **Requirements Analysis:** The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. In other words, we must find out what the users want from the database.
The Entity-Relationship Model

**Database design tools:** Design tools are available from RDBMS vendors as well as third-party vendors. Sybase and Oracle, in particular, have comprehensive sets of design and analysis tools. See the following URL for details on Sybase’s tools: [http://www.sybase.com/products/application_tools](http://www.sybase.com/products/application_tools) The following provides details on Oracle’s tools: [http://www.oracle.com/tools](http://www.oracle.com/tools)

This is usually an informal process that involves discussions with user groups, a study of the current operating environment and how it is expected to change, analysis of any available documentation on existing applications that are expected to be replaced or complemented by the database, and so on. Several methodologies have been proposed for organizing and presenting the information gathered in this step, and some automated tools have been developed to support this process.

(2) **Conceptual Database Design:** The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints that are known to hold over this data. This step is often carried out using the ER model, or a similar high-level data model, and is discussed in the rest of this chapter.

(3) **Logical Database Design:** We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model of the chosen DBMS. We will only consider relational DBMSs, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema. We discuss this step in detail in Chapter 3; the result is a conceptual schema, sometimes called the logical schema, in the relational data model.

### 2.1.1 Beyond the ER Model

ER modeling is sometimes regarded as a complete approach to designing a logical database schema. This is incorrect because the ER diagram is just an approximate description of the data, constructed through a very subjective evaluation of the information collected during requirements analysis. A more careful analysis can often refine the logical schema obtained at the end of Step 3. Once we have a good logical schema, we must consider performance criteria and design the physical schema. Finally, we must address security issues and ensure that users are able to access the data they need, but not data that we wish to hide from them. The remaining three steps of database design are briefly described below: ¹

¹This material can be omitted on a first reading of this chapter without loss of continuity.
(4) Schema Refinement: The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems, and to refine it. In contrast to the requirements analysis and conceptual design steps, which are essentially subjective, schema refinement can be guided by some elegant and powerful theory. We discuss the theory of normalizing relations—restructuring them to ensure some desirable properties—in Chapter 15.

(5) Physical Database Design: In this step we must consider typical expected workloads that our database must support and further refine the database design to ensure that it meets desired performance criteria. This step may simply involve building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema obtained from the earlier design steps. We discuss physical design and database tuning in Chapter 16.

(6) Security Design: In this step, we identify different user groups and different roles played by various users (e.g., the development team for a product, the customer support representatives, the product manager). For each role and user group, we must identify the parts of the database that they must be able to access and the parts of the database that they should not be allowed to access, and take steps to ensure that they can access only the necessary parts. A DBMS provides several mechanisms to assist in this step, and we discuss this in Chapter 17.

In general, our division of the design process into steps should be seen as a classification of the kinds of steps involved in design. Realistically, although we might begin with the six step process outlined here, a complete database design will probably require a subsequent tuning phase in which all six kinds of design steps are interleaved and repeated until the design is satisfactory. Further, we have omitted the important steps of implementing the database design, and designing and implementing the application layers that run on top of the DBMS. In practice, of course, these additional steps can lead to a rethinking of the basic database design.

The concepts and techniques that underlie a relational DBMS are clearly useful to someone who wants to implement or maintain the internals of a database system. However, it is important to recognize that serious users and DBAs must also know how a DBMS works. A good understanding of database system internals is essential for a user who wishes to take full advantage of a DBMS and design a good database; this is especially true of physical design and database tuning.

2.2 ENTITIES, ATTRIBUTES, AND ENTITY SETS

An entity is an object in the real world that is distinguishable from other objects. Examples include the following: the Green Dragonzord toy, the toy department, the manager of the toy department, the home address of the manager of the toy depart-
It is often useful to identify a collection of similar entities. Such a collection is called an entity set. Note that entity sets need not be disjoint; the collection of toy department employees and the collection of appliance department employees may both contain employee John Doe (who happens to work in both departments). We could also define an entity set called Employees that contains both the toy and appliance department employee sets.

An entity is described using a set of attributes. All entities in a given entity set have the same attributes; this is essentially what we mean by similar. (This statement is an oversimplification, as we will see when we discuss inheritance hierarchies in Section 2.4.4, but it suffices for now and highlights the main idea.) Our choice of attributes reflects the level of detail at which we wish to represent information about entities. For example, the Employees entity set could use name, social security number (ssn), and parking lot (lot) as attributes. In this case we will store the name, social security number, and lot number for each employee. However, we will not store, say, an employee’s address (or gender or age).

For each attribute associated with an entity set, we must identify a domain of possible values. For example, the domain associated with the attribute name of Employees might be the set of 20-character strings. As another example, if the company rates employees on a scale of 1 to 10 and stores ratings in a field called rating, the associated domain consists of integers 1 through 10. Further, for each entity set, we choose a key. A key is a minimal set of attributes whose values uniquely identify an entity in the set. There could be more than one candidate key; if so, we designate one of them as the primary key. For now we will assume that each entity set contains at least one set of attributes that uniquely identifies an entity in the entity set; that is, the set of attributes contains a key. We will revisit this point in Section 2.4.3.

The Employees entity set with attributes ssn, name, and lot is shown in Figure 2.1. An entity set is represented by a rectangle, and an attribute is represented by an oval. Each attribute in the primary key is underlined. The domain information could be listed along with the attribute name, but we omit this to keep the figures compact. The key is ssn.

2.3 RELATIONSHIPS AND RELATIONSHIP SETS

A relationship is an association among two or more entities. For example, we may have the relationship that Attishoo works in the pharmacy department. As with entities, we may wish to collect a set of similar relationships into a relationship set.

\footnote{To avoid confusion, we will assume that attribute names do not repeat across entity sets. This is not a real limitation because we can always use the entity set name to resolve ambiguities if the same attribute name is used in more than one entity set.}
A relationship set can be thought of as a set of $n$-tuples:

$$\{(e_1, \ldots, e_n) \mid e_1 \in E_1, \ldots, e_n \in E_n\}$$

Each $n$-tuple denotes a relationship involving $n$ entities $e_1$ through $e_n$, where entity $e_i$ is in entity set $E_i$. In Figure 2.2 we show the relationship set Works_In, in which each relationship indicates a department in which an employee works. Note that several relationship sets might involve the same entity sets. For example, we could also have a Manages relationship set involving Employees and Departments.

A relationship can also have **descriptive attributes**. Descriptive attributes are used to record information about the relationship, rather than about any one of the participating entities; for example, we may wish to record that Attishoo works in the pharmacy department as of January 1991. This information is captured in Figure 2.2 by adding an attribute, \textit{since}, to Works_In. A relationship must be uniquely identified by the participating entities, without reference to the descriptive attributes. In the Works_In relationship set, for example, each Works_In relationship must be uniquely identified by the combination of employee \textit{ssn} and department \textit{did}. Thus, for a given employee-department pair, we cannot have more than one associated \textit{since} value.

An **instance** of a relationship set is a set of relationships. Intuitively, an instance can be thought of as a ‘snapshot’ of the relationship set at some instant in time. An instance of the Works_In relationship set is shown in Figure 2.3. Each Employees entity is denoted by its \textit{ssn}, and each Departments entity is denoted by its \textit{did}, for simplicity.
The *since* value is shown beside each relationship. (The ‘many-to-many’ and ‘total participation’ comments in the figure will be discussed later, when we discuss integrity constraints.)

![Figure 2.3 An Instance of the Works_In Relationship Set](image)

As another example of an ER diagram, suppose that each department has offices in several locations and we want to record the locations at which each employee works. This relationship is **ternary** because we must record an association between an employee, a department, and a location. The ER diagram for this variant of Works_In, which we call Works_In2, is shown in Figure 2.4.

![Figure 2.4 A Ternary Relationship Set](image)

The entity sets that participate in a relationship set need not be distinct; sometimes a relationship might involve two entities in the same entity set. For example, consider the Reports_To relationship set that is shown in Figure 2.5. Since employees report to other employees, every relationship in Reports_To is of the form \((emp_1, emp_2)\),
where both $emp_1$ and $emp_2$ are entities in Employees. However, they play different roles: $emp_1$ reports to the managing employee $emp_2$, which is reflected in the role indicators $supervisor$ and $subordinate$ in Figure 2.5. If an entity set plays more than one role, the role indicator concatenated with an attribute name from the entity set gives us a unique name for each attribute in the relationship set. For example, the Reports-To relationship set has attributes corresponding to the $ssn$ of the supervisor and the $ssn$ of the subordinate, and the names of these attributes are $supervisor_ssn$ and $subordinate_ssn$.

![Figure 2.5](image)

**Figure 2.5** The Reports_To Relationship Set

2.4 ADDITIONAL FEATURES OF THE ER MODEL

We now look at some of the constructs in the ER model that allow us to describe some subtle properties of the data. The expressiveness of the ER model is a big reason for its widespread use.

2.4.1 Key Constraints

Consider the Works_In relationship shown in Figure 2.2. An employee can work in several departments, and a department can have several employees, as illustrated in the Works_In instance shown in Figure 2.3. Employee 231-31-5368 has worked in Department 51 since 3/3/93 and in Department 56 since 2/2/92. Department 51 has two employees.

Now consider another relationship set called Manages between the Employees and Departments entity sets such that each department has at most one manager, although a single employee is allowed to manage more than one department. The restriction that each department has at most one manager is an example of a key constraint, and it implies that each Departments entity appears in at most one Manages relationship.
in any allowable instance of Manages. This restriction is indicated in the ER diagram of Figure 2.6 by using an arrow from Departments to Manages. Intuitively, the arrow states that given a Departments entity, we can uniquely determine the Manages relationship in which it appears.

![ER Diagram](image)

**Figure 2.6** Key Constraint on Manages

An instance of the Manages relationship set is shown in Figure 2.7. While this is also a potential instance for the Works_In relationship set, the instance of Works_In shown in Figure 2.3 violates the key constraint on Manages.

![Instance Diagram](image)

**Figure 2.7** An Instance of the Manages Relationship Set

A relationship set like Manages is sometimes said to be **one-to-many**, to indicate that one employee can be associated with many departments (in the capacity of a manager), whereas each department can be associated with at most one employee as its manager. In contrast, the Works_In relationship set, in which an employee is allowed to work in several departments and a department is allowed to have several employees, is said to be **many-to-many**.
If we add the restriction that each employee can manage at most one department to the Manages relationship set, which would be indicated by adding an arrow from Employees to Manages in Figure 2.6, we have a one-to-one relationship set.

**Key Constraints for Ternary Relationships**

We can extend this convention—and the underlying key constraint concept—to relationship sets involving three or more entity sets: If an entity set E has a key constraint in a relationship set R, each entity in an instance of E appears in at most one relationship in (a corresponding instance of) R. To indicate a key constraint on entity set E in relationship set R, we draw an arrow from E to R.

In Figure 2.8, we show a ternary relationship with key constraints. Each employee works in at most one department, and at a single location. An instance of the Works_In3 relationship set is shown in Figure 2.9. Notice that each department can be associated with several employees and locations, and each location can be associated with several departments and employees; however, each employee is associated with a single department and location.

![Figure 2.8 A Ternary Relationship Set with Key Constraints](image)

2.4.2 Participation Constraints

The key constraint on Manages tells us that a department has at most one manager. A natural question to ask is whether every department has a manager. Let us say that every department is required to have a manager. This requirement is an example of a participation constraint: the participation of the entity set Departments in the relationship set Manages is said to be total. A participation that is not total is said to be partial. As an example, the participation of the entity set Employees in Manages is partial, since not every employee gets to manage a department.
Revisiting the Works_In relationship set, it is natural to expect that each employee works in at least one department and that each department has at least one employee. This means that the participation of both Employees and Departments in Works_In is total. The ER diagram in Figure 2.10 shows both the Manages and Works_In relationship sets and all the given constraints. If the participation of an entity set in a relationship set is total, the two are connected by a thick line; independently, the presence of an arrow indicates a key constraint. The instances of Works_In and Manages shown in Figures 2.3 and 2.7 satisfy all the constraints in Figure 2.10.

### 2.4.3 Weak Entities

Thus far, we have assumed that the attributes associated with an entity set include a key. This assumption does not always hold. For example, suppose that employees can purchase insurance policies to cover their dependents. We wish to record information about policies, including who is covered by each policy, but this information is really our only interest in the dependents of an employee. If an employee quits, any policy owned by the employee is terminated and we want to delete all the relevant policy and dependent information from the database.

We might choose to identify a dependent by name alone in this situation, since it is reasonable to expect that the dependents of a given employee have different names. Thus the attributes of the Dependents entity set might be pname and age. The attribute pname does not identify a dependent uniquely. Recall that the key for Employees is
Dependents is an example of a weak entity set. A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity, which is called the identifying owner.

The following restrictions must hold:

- The owner entity set and the weak entity set must participate in a one-to-many relationship set (one owner entity is associated with one or more weak entities, but each weak entity has a single owner). This relationship set is called the identifying relationship set of the weak entity set.

- The weak entity set must have total participation in the identifying relationship set.

For example, a Dependents entity can be identified uniquely only if we take the key of the owning Employees entity and the pname of the Dependents entity. The set of attributes of a weak entity set that uniquely identify a weak entity for a given owner entity is called a partial key of the weak entity set. In our example pname is a partial key for Dependents.

The Dependents weak entity set and its relationship to Employees is shown in Figure 2.11. The total participation of Dependents in Policy is indicated by linking them...
The Entity-Relationship Model

with a dark line. The arrow from Dependents to Policy indicates that each Dependents entity appears in at most one (indeed, exactly one, because of the participation constraint) Policy relationship. To underscore the fact that Dependents is a weak entity and Policy is its identifying relationship, we draw both with dark lines. To indicate that \textit{pname} is a partial key for Dependents, we underline it using a broken line. This means that there may well be two dependents with the same \textit{pname} value.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{weak_entity_set.png}
\caption{A Weak Entity Set}
\end{figure}

\subsection{Class Hierarchies}

Sometimes it is natural to classify the entities in an entity set into subclasses. For example, we might want to talk about an Hourly\_Emps entity set and a Contract\_Emps entity set to distinguish the basis on which they are paid. We might have attributes \textit{hours\_worked} and \textit{hourly\_wage} defined for Hourly\_Emps and an attribute \textit{contract\_id} defined for Contract\_Emps.

We want the semantics that every entity in one of these sets is also an Employees entity, and as such must have all of the attributes of Employees defined. Thus, the attributes defined for an Hourly\_Emps entity are the attributes for Employees plus Hourly\_Emps. We say that the attributes for the entity set Employees are \textbf{inherited} by the entity set Hourly\_Emps, and that Hourly\_Emps \textbf{ISA} (read \textit{is a}) Employees. In addition—and in contrast to class hierarchies in programming languages such as C++—there is a constraint on queries over instances of these entity sets: A query that asks for all Employees entities must consider all Hourly\_Emps and Contract\_Emps entities as well. Figure 2.12 illustrates the class hierarchy.

The entity set Employees may also be classified using a different criterion. For example, we might identify a subset of employees as Senior\_Emps. We can modify Figure 2.12 to reflect this change by adding a second ISA node as a child of Employees and making Senior\_Emps a child of this node. Each of these entity sets might be classified further, creating a multilevel ISA hierarchy.

A class hierarchy can be viewed in one of two ways:
Employees is **specialized** into subclasses. Specialization is the process of identifying subsets of an entity set (the **superclass**) that share some distinguishing characteristic. Typically the superclass is defined first, the subclasses are defined next, and subclass-specific attributes and relationship sets are then added.

- Hourly_Emps and Contract_Emps are **generalized** by Employees. As another example, two entity sets Motorboats and Cars may be generalized into an entity set Motor_Vehicles. Generalization consists of identifying some common characteristics of a collection of entity sets and creating a new entity set that contains entities possessing these common characteristics. Typically the subclasses are defined first, the superclass is defined next, and any relationship sets that involve the superclass are then defined.

We can specify two kinds of constraints with respect to ISA hierarchies, namely, overlap and covering constraints. **Overlap constraints** determine whether two subclasses are allowed to contain the same entity. For example, can Attishoo be both an Hourly_Emps entity and a Contract_Emps entity? Intuitively, no. Can he be both a Contract_Emps entity and a Senior_Emps entity? Intuitively, yes. We denote this by writing ‘Contract_Emps OVERLAPS Senior_Emps.’ In the absence of such a statement, we assume by default that entity sets are constrained to have no overlap.

**Covering constraints** determine whether the entities in the subclasses collectively include all entities in the superclass. For example, does every Employees entity have to belong to one of its subclasses? Intuitively, no. Does every Motor_Vehicles entity have to be either a Motorboats entity or a Cars entity? Intuitively, yes; a characteristic property of generalization hierarchies is that every instance of a superclass is an instance of a subclass. We denote this by writing ‘Motorboats AND Cars COVER
Motor_Vehicles.’ In the absence of such a statement, we assume by default that there is no covering constraint; we can have motor vehicles that are not motorboats or cars.

There are two basic reasons for identifying subclasses (by specialization or generalization):

1. We might want to add descriptive attributes that make sense only for the entities in a subclass. For example, hourly_wages does not make sense for a Contract_Emps entity, whose pay is determined by an individual contract.

2. We might want to identify the set of entities that participate in some relationship. For example, we might wish to define the Manages relationship so that the participating entity sets are Senior_Emps and Departments, to ensure that only senior employees can be managers. As another example, Motorboats and Cars may have different descriptive attributes (say, tonnage and number of doors), but as Motor_Vehicles entities, they must be licensed. The licensing information can be captured by a Licensed_To relationship between Motor_Vehicles and an entity set called Owners.

2.4.5 Aggregation

As we have defined it thus far, a relationship set is an association between entity sets. Sometimes we have to model a relationship between a collection of entities and relationships. Suppose that we have an entity set called Projects and that each Projects entity is sponsored by one or more departments. The Sponsors relationship set captures this information. A department that sponsors a project might assign employees to monitor the sponsorship. Intuitively, Monitors should be a relationship set that associates a Sponsors relationship (rather than a Projects or Departments entity) with an Employees entity. However, we have defined relationships to associate two or more entities.

In order to define a relationship set such as Monitors, we introduce a new feature of the ER model, called aggregation. Aggregation allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship set. This is illustrated in Figure 2.13, with a dashed box around Sponsors (and its participating entity sets) used to denote aggregation. This effectively allows us to treat Sponsors as an entity set for purposes of defining the Monitors relationship set.

When should we use aggregation? Intuitively, we use it when we need to express a relationship among relationships. But can’t we express relationships involving other relationships without using aggregation? In our example, why not make Sponsors a ternary relationship? The answer is that there are really two distinct relationships, Sponsors and Monitors, each possibly with attributes of its own. For instance, the
Monitors relationship has an attribute \textit{until} that records the date until when the employee is appointed as the sponsorship monitor. Compare this attribute with the attribute \textit{since} of Sponsors, which is the date when the sponsorship took effect. The use of aggregation versus a ternary relationship may also be guided by certain integrity constraints, as explained in Section 2.5.4.

2.5 CONCEPTUAL DATABASE DESIGN WITH THE ER MODEL

Developing an ER diagram presents several choices, including the following:

- Should a concept be modeled as an entity or an attribute?
- Should a concept be modeled as an entity or a relationship?
- What are the relationship sets and their participating entity sets? Should we use binary or ternary relationships?
- Should we use aggregation?

We now discuss the issues involved in making these choices.
### 2.5.1 Entity versus Attribute

While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modeled as an attribute or as an entity set (and related to the first entity set using a relationship set). For example, consider adding address information to the Employees entity set. One option is to use an attribute *address*. This option is appropriate if we need to record only one address per employee, and it suffices to think of an address as a string. An alternative is to create an entity set called Addresses and to record associations between employees and addresses using a relationship (say, Has_Address). This more complex alternative is necessary in two situations:

- We have to record more than one address for an employee.
- We want to capture the structure of an address in our ER diagram. For example, we might break down an address into city, state, country, and Zip code, in addition to a string for street information. By representing an address as an entity with these attributes, we can support queries such as “Find all employees with an address in Madison, WI.”

For another example of when to model a concept as an entity set rather than as an attribute, consider the relationship set (called Works_In2) shown in Figure 2.14.

![Figure 2.14 The Works_In2 Relationship Set](image)

It differs from the Works_In relationship set of Figure 2.2 only in that it has attributes *from* and *to*, instead of *since*. Intuitively, it records the interval during which an employee works for a department. Now suppose that it is possible for an employee to work in a given department over more than one period.

This possibility is ruled out by the ER diagram’s semantics. The problem is that we want to record several values for the descriptive attributes for each instance of the Works_In2 relationship. (This situation is analogous to wanting to record several addresses for each employee.) We can address this problem by introducing an entity set called, say, Duration, with attributes *from* and *to*, as shown in Figure 2.15.
In some versions of the ER model, attributes are allowed to take on sets as values. Given this feature, we could make Duration an attribute of Works_In, rather than an entity set; associated with each Works_In relationship, we would have a set of intervals. This approach is perhaps more intuitive than modeling Duration as an entity set. Nonetheless, when such set-valued attributes are translated into the relational model, which does not support set-valued attributes, the resulting relational schema is very similar to what we get by regarding Duration as an entity set.

### 2.5.2 Entity versus Relationship

Consider the relationship set called Manages in Figure 2.6. Suppose that each department manager is given a discretionary budget ($dbudget$), as shown in Figure 2.16, in which we have also renamed the relationship set to Manages2.

There is at most one employee managing a department, but a given employee could manage several departments; we store the starting date and discretionary budget for each manager-department pair. This approach is natural if we assume that a manager receives a separate discretionary budget for each department that he or she manages.
But what if the discretionary budget is a sum that covers all departments managed by that employee? In this case each Manages2 relationship that involves a given employee will have the same value in the \emph{dbudget} field. In general such redundancy could be significant and could cause a variety of problems. (We discuss redundancy and its attendant problems in Chapter 15.) Another problem with this design is that it is misleading.

We can address these problems by associating \emph{dbudget} with the appointment of the employee as manager of a group of departments. In this approach, we model the appointment as an entity set, say Mgr_Appt, and use a ternary relationship, say Manages3, to relate a manager, an appointment, and a department. The details of an appointment (such as the discretionary budget) are not repeated for each department that is included in the appointment now, although there is still one Manages3 relationship instance per such department. Further, note that each department has at most one manager, as before, because of the key constraint. This approach is illustrated in Figure 2.17.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image.png}
\caption{Entity Set versus Relationship}
\end{figure}

\subsection{Binary versus Ternary Relationships *}

Consider the ER diagram shown in Figure 2.18. It models a situation in which an employee can own several policies, each policy can be owned by several employees, and each dependent can be covered by several policies.

Suppose that we have the following additional requirements:

- A policy cannot be owned jointly by two or more employees.
- Every policy must be owned by some employee.
Dependents is a weak entity set, and each dependent entity is uniquely identified by taking \textit{name} in conjunction with the \textit{policyid} of a policy entity (which, intuitively, covers the given dependent).

The first requirement suggests that we impose a key constraint on Policies with respect to Covers, but this constraint has the unintended side effect that a policy can cover only one dependent. The second requirement suggests that we impose a total participation constraint on Policies. This solution is acceptable if each policy covers at least one dependent. The third requirement forces us to introduce an identifying relationship that is binary (in our version of ER diagrams, although there are versions in which this is not the case).

Even ignoring the third point above, the best way to model this situation is to use two binary relationships, as shown in Figure 2.19.

This example really had two relationships involving Policies, and our attempt to use a single ternary relationship (Figure 2.18) was inappropriate. There are situations, however, where a relationship inherently associates more than two entities. We have seen such an example in Figure 2.4 and also Figures 2.15 and 2.17.

As a good example of a ternary relationship, consider entity sets Parts, Suppliers, and Departments, and a relationship set Contracts (with descriptive attribute \textit{qty}) that involves all of them. A contract specifies that a supplier will supply (some quantity of) a part to a department. This relationship cannot be adequately captured by a collection of binary relationships (without the use of aggregation). With binary relationships, we can denote that a supplier ‘can supply’ certain parts, that a department ‘needs’ some
parts, or that a department ‘deals with’ a certain supplier. No combination of these relationships expresses the meaning of a contract adequately, for at least two reasons:

- The facts that supplier S can supply part P, that department D needs part P, and that D will buy from S do not necessarily imply that department D indeed buys part P from supplier S!
- We cannot represent the qty attribute of a contract cleanly.

### 2.5.4 Aggregation versus Ternary Relationships *

As we noted in Section 2.4.5, the choice between using aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a relationship set to an entity set (or second relationship set). The choice may also be guided by certain integrity constraints that we want to express. For example, consider the ER diagram shown in Figure 2.13. According to this diagram, a project can be sponsored by any number of departments, a department can sponsor one or more projects, and each sponsorship is monitored by one or more employees. If we don’t need to record the until attribute of Monitors, then we might reasonably use a ternary relationship, say, Sponsors2, as shown in Figure 2.20.

Consider the constraint that each sponsorship (of a project by a department) be monitored by at most one employee. We cannot express this constraint in terms of the Sponsors2 relationship set. On the other hand, we can easily express the constraint by drawing an arrow from the aggregated relationship Sponsors to the relationship...
Monitors in Figure 2.13. Thus, the presence of such a constraint serves as another reason for using aggregation rather than a ternary relationship set.

2.6 CONCEPTUAL DESIGN FOR LARGE ENTERPRISES *

We have thus far concentrated on the constructs available in the ER model for describing various application concepts and relationships. The process of conceptual design consists of more than just describing small fragments of the application in terms of ER diagrams. For a large enterprise, the design may require the efforts of more than one designer and span data and application code used by a number of user groups. Using a high-level, semantic data model such as ER diagrams for conceptual design in such an environment offers the additional advantage that the high-level design can be diagrammatically represented and is easily understood by the many people who must provide input to the design process.

An important aspect of the design process is the methodology used to structure the development of the overall design and to ensure that the design takes into account all user requirements and is consistent. The usual approach is that the requirements of various user groups are considered, any conflicting requirements are somehow resolved, and a single set of global requirements is generated at the end of the requirements analysis phase. Generating a single set of global requirements is a difficult task, but it allows the conceptual design phase to proceed with the development of a logical schema that spans all the data and applications throughout the enterprise.
An alternative approach is to develop separate conceptual schemas for different user groups and to then integrate these conceptual schemas. To integrate multiple conceptual schemas, we must establish correspondences between entities, relationships, and attributes, and we must resolve numerous kinds of conflicts (e.g., naming conflicts, domain mismatches, differences in measurement units). This task is difficult in its own right. In some situations schema integration cannot be avoided—for example, when one organization merges with another, existing databases may have to be integrated. Schema integration is also increasing in importance as users demand access to heterogeneous data sources, often maintained by different organizations.

2.7 POINTS TO REVIEW

- Database design has six steps: requirements analysis, conceptual database design, logical database design, schema refinement, physical database design, and security design. Conceptual design should produce a high-level description of the data, and the entity-relationship (ER) data model provides a graphical approach to this design phase. (Section 2.1)

- In the ER model, a real-world object is represented as an entity. An entity set is a collection of structurally identical entities. Entities are described using attributes. Each entity set has a distinguished set of attributes called a key that can be used to uniquely identify each entity. (Section 2.2)

- A relationship is an association between two or more entities. A relationship set is a collection of relationships that relate entities from the same entity sets. A relationship can also have descriptive attributes. (Section 2.3)

- A key constraint between an entity set S and a relationship set restricts instances of the relationship set by requiring that each entity of S participate in at most one relationship. A participation constraint between an entity set S and a relationship set restricts instances of the relationship set by requiring that each entity of S participate in at least one relationship. The identity and existence of a weak entity depends on the identity and existence of another (owner) entity. Class hierarchies organize structurally similar entities through inheritance into sub- and super-classes. Aggregation conceptually transforms a relationship set into an entity set such that the resulting construct can be related to other entity sets. (Section 2.4)

- Development of an ER diagram involves important modeling decisions. A thorough understanding of the problem being modeled is necessary to decide whether to use an attribute or an entity set, an entity or a relationship set, a binary or ternary relationship, or aggregation. (Section 2.5)

- Conceptual design for large enterprises is especially challenging because data from many sources, managed by many groups, is involved. (Section 2.6)
EXERCISES

Exercise 2.1 Explain the following terms briefly: attribute, domain, entity, relationship, entity set, relationship set, one-to-many relationship, many-to-many relationship, participation constraint, overlap constraint, covering constraint, weak entity set, aggregation, and role indicator.

Exercise 2.2 A university database contains information about professors (identified by social security number, or SSN) and courses (identified by courseid). Professors teach courses; each of the following situations concerns the Teaches relationship set. For each situation, draw an ER diagram that describes it (assuming that no further constraints hold).

1. Professors can teach the same course in several semesters, and each offering must be recorded.
2. Professors can teach the same course in several semesters, and only the most recent such offering needs to be recorded. (Assume this condition applies in all subsequent questions.)
3. Every professor must teach some course.
4. Every professor teaches exactly one course (no more, no less).
5. Every professor teaches exactly one course (no more, no less), and every course must be taught by some professor.
6. Now suppose that certain courses can be taught by a team of professors jointly, but it is possible that no one professor in a team can teach the course. Model this situation, introducing additional entity sets and relationship sets if necessary.

Exercise 2.3 Consider the following information about a university database:

- Professors have an SSN, a name, an age, a rank, and a research specialty.
- Projects have a project number, a sponsor name (e.g., NSF), a starting date, an ending date, and a budget.
- Graduate students have an SSN, a name, an age, and a degree program (e.g., M.S. or Ph.D.).
- Each project is managed by one professor (known as the project’s principal investigator).
- Each project is worked on by one or more professors (known as the project’s co-investigators).
- Professors can manage and/or work on multiple projects.
- Each project is worked on by one or more graduate students (known as the project’s research assistants).
- When graduate students work on a project, a professor must supervise their work on the project. Graduate students can work on multiple projects, in which case they will have a (potentially different) supervisor for each one.
- Departments have a department number, a department name, and a main office.
- Departments have a professor (known as the chairman) who runs the department.
- Professors work in one or more departments, and for each department that they work in, a time percentage is associated with their job.
Graduate students have one major department in which they are working on their degree.

Each graduate student has another, more senior graduate student (known as a student advisor) who advises him or her on what courses to take.

Design and draw an ER diagram that captures the information about the university. Use only the basic ER model here, that is, entities, relationships, and attributes. Be sure to indicate any key and participation constraints.

**Exercise 2.4** A company database needs to store information about employees (identified by ssn, with salary and phone as attributes); departments (identified by dno, with dname and budget as attributes); and children of employees (with name and age as attributes). Employees work in departments; each department is managed by an employee; a child must be identified uniquely by name when the parent (who is an employee; assume that only one parent works for the company) is known. We are not interested in information about a child once the parent leaves the company.

Draw an ER diagram that captures this information.

**Exercise 2.5** Notown Records has decided to store information about musicians who perform on its albums (as well as other company data) in a database. The company has wisely chosen to hire you as a database designer (at your usual consulting fee of $2,500/day).

- Each musician that records at Notown has an SSN, a name, an address, and a phone number. Poorly paid musicians often share the same address, and no address has more than one phone.
- Each instrument that is used in songs recorded at Notown has a name (e.g., guitar, synthesizer, flute) and a musical key (e.g., C, B-flat, E-flat).
- Each album that is recorded on the Notown label has a title, a copyright date, a format (e.g., CD or MC), and an album identifier.
- Each song recorded at Notown has a title and an author.
- Each musician may play several instruments, and a given instrument may be played by several musicians.
- Each album has a number of songs on it, but no song may appear on more than one album.
- Each song is performed by one or more musicians, and a musician may perform a number of songs.
- Each album has exactly one musician who acts as its producer. A musician may produce several albums, of course.

Design a conceptual schema for Notown and draw an ER diagram for your schema. The following information describes the situation that the Notown database must model. Be sure to indicate all key and cardinality constraints and any assumptions that you make. Identify any constraints that you are unable to capture in the ER diagram and briefly explain why you could not express them.
Exercise 2.6  Computer Sciences Department frequent fliers have been complaining to Dane County Airport officials about the poor organization at the airport. As a result, the officials have decided that all information related to the airport should be organized using a DBMS, and you've been hired to design the database. Your first task is to organize the information about all the airplanes that are stationed and maintained at the airport. The relevant information is as follows:

- Every airplane has a registration number, and each airplane is of a specific model.
- The airport accommodates a number of airplane models, and each model is identified by a model number (e.g., DC-10) and has a capacity and a weight.
- A number of technicians work at the airport. You need to store the name, SSN, address, phone number, and salary of each technician.
- Each technician is an expert on one or more plane model(s), and his or her expertise may overlap with that of other technicians. This information about technicians must also be recorded.
- Traffic controllers must have an annual medical examination. For each traffic controller, you must store the date of the most recent exam.
- All airport employees (including technicians) belong to a union. You must store the union membership number of each employee. You can assume that each employee is uniquely identified by the social security number.
- The airport has a number of tests that are used periodically to ensure that airplanes are still airworthy. Each test has a Federal Aviation Administration (FAA) test number, a name, and a maximum possible score.
- The FAA requires the airport to keep track of each time that a given airplane is tested by a given technician using a given test. For each testing event, the information needed is the date, the number of hours the technician spent doing the test, and the score that the airplane received on the test.

1. Draw an ER diagram for the airport database. Be sure to indicate the various attributes of each entity and relationship set; also specify the key and participation constraints for each relationship set. Specify any necessary overlap and covering constraints as well (in English).

2. The FAA passes a regulation that tests on a plane must be conducted by a technician who is an expert on that model. How would you express this constraint in the ER diagram? If you cannot express it, explain briefly.

Exercise 2.7  The Prescriptions-R-X chain of pharmacies has offered to give you a free lifetime supply of medicines if you design its database. Given the rising cost of health care, you agree. Here's the information that you gather:

- Patients are identified by an SSN, and their names, addresses, and ages must be recorded.
- Doctors are identified by an SSN. For each doctor, the name, specialty, and years of experience must be recorded.
- Each pharmaceutical company is identified by name and has a phone number.
For each drug, the trade name and formula must be recorded. Each drug is sold by a given pharmaceutical company, and the trade name identifies a drug uniquely from among the products of that company. If a pharmaceutical company is deleted, you need not keep track of its products any longer.

Each pharmacy has a name, address, and phone number.

Every patient has a primary physician. Every doctor has at least one patient.

Each pharmacy sells several drugs and has a price for each. A drug could be sold at several pharmacies, and the price could vary from one pharmacy to another.

Doctors prescribe drugs for patients. A doctor could prescribe one or more drugs for several patients, and a patient could obtain prescriptions from several doctors. Each prescription has a date and a quantity associated with it. You can assume that if a doctor prescribes the same drug for the same patient more than once, only the last such prescription needs to be stored.

Pharmaceutical companies have long-term contracts with pharmacies. A pharmaceutical company can contract with several pharmacies, and a pharmacy can contract with several pharmaceutical companies. For each contract, you have to store a start date, an end date, and the text of the contract.

Pharmacies appoint a supervisor for each contract. There must always be a supervisor for each contract, but the contract supervisor can change over the lifetime of the contract.

1. Draw an ER diagram that captures the above information. Identify any constraints that are not captured by the ER diagram.

2. How would your design change if each drug must be sold at a fixed price by all pharmacies?

3. How would your design change if the design requirements change as follows: If a doctor prescribes the same drug for the same patient more than once, several such prescriptions may have to be stored.

Exercise 2.8 Although you always wanted to be an artist, you ended up being an expert on databases because you love to cook data and you somehow confused ‘database’ with ‘data baste.’ Your old love is still there, however, so you set up a database company, ArtBase, that builds a product for art galleries. The core of this product is a database with a schema that captures all the information that galleries need to maintain. Galleries keep information about artists, their names (which are unique), birthplaces, age, and style of art. For each piece of artwork, the artist, the year it was made, its unique title, its type of art (e.g., painting, lithograph, sculpture, photograph), and its price must be stored. Pieces of artwork are also classified into groups of various kinds, for example, portraits, still lifes, works by Picasso, or works of the 19th century; a given piece may belong to more than one group. Each group is identified by a name (like those above) that describes the group. Finally, galleries keep information about customers. For each customer, galleries keep their unique name, address, total amount of dollars they have spent in the gallery (very important!), and the artists and groups of art that each customer tends to like.

Draw the ER diagram for the database.
BIBLIOGRAPHIC NOTES

Several books provide a good treatment of conceptual design; these include [52] (which also contains a survey of commercial database design tools) and [641].

The ER model was proposed by Chen [145], and extensions have been proposed in a number of subsequent papers. Generalization and aggregation were introduced in [604]. [330] and [514] contain good surveys of semantic data models. Dynamic and temporal aspects of semantic data models are discussed in [658].

[642] discusses a design methodology based on developing an ER diagram and then translating to the relational model. Markowitz considers referential integrity in the context of ER to relational mapping and discusses the support provided in some commercial systems (as of that date) in [446, 447].

The entity-relationship conference proceedings contain numerous papers on conceptual design, with an emphasis on the ER model, for example, [609].

View integration is discussed in several papers, including [84, 118, 153, 207, 465, 480, 479, 596, 608, 657]. [53] is a survey of several integration approaches.
Codd proposed the relational data model in 1970. At that time most database systems were based on one of two older data models (the hierarchical model and the network model); the relational model revolutionized the database field and largely supplanted these earlier models. Prototype relational database management systems were developed in pioneering research projects at IBM and UC-Berkeley by the mid-70s, and several vendors were offering relational database products shortly thereafter. Today, the relational model is by far the dominant data model and is the foundation for the leading DBMS products, including IBM’s DB2 family, Informix, Oracle, Sybase, Microsoft’s Access and SQLServer, FoxBase, and Paradox. Relational database systems are ubiquitous in the marketplace and represent a multibillion dollar industry.

The relational model is very simple and elegant; a database is a collection of one or more relations, where each relation is a table with rows and columns. This simple tabular representation enables even novice users to understand the contents of a database, and it permits the use of simple, high-level languages to query the data. The major advantages of the relational model over the older data models are its simple data representation and the ease with which even complex queries can be expressed.

This chapter introduces the relational model and covers the following issues:

- How is data represented?
- What kinds of integrity constraints can be expressed?
- How can data be created and modified?
- How can data be manipulated and queried?
- How do we obtain a database design in the relational model?
- How are logical and physical data independence achieved?
**SQL:** It was the query language of the pioneering System-R relational DBMS developed at IBM. Over the years, SQL has become the most widely used language for creating, manipulating, and querying relational DBMSs. Since many vendors offer SQL products, there is a need for a standard that defines ‘official SQL.’ The existence of a standard allows users to measure a given vendor’s version of SQL for completeness. It also allows users to distinguish SQL features that are specific to one product from those that are standard; an application that relies on non-standard features is less portable.

The first SQL standard was developed in 1986 by the American National Standards Institute (ANSI), and was called SQL-86. There was a minor revision in 1989 called SQL-89, and a major revision in 1992 called SQL-92. The International Standards Organization (ISO) collaborated with ANSI to develop SQL-92. Most commercial DBMSs currently support SQL-92. An exciting development is the imminent approval of SQL:1999, a major extension of SQL-92. While the coverage of SQL in this book is based upon SQL-92, we will cover the main extensions of SQL:1999 as well.

While we concentrate on the underlying concepts, we also introduce the **Data Definition Language (DDL)** features of SQL-92, the standard language for creating, manipulating, and querying data in a relational DBMS. This allows us to ground the discussion firmly in terms of real database systems.

We discuss the concept of a relation in Section 3.1 and show how to create relations using the SQL language. An important component of a data model is the set of constructs it provides for specifying conditions that must be satisfied by the data. Such conditions, called **integrity constraints** (ICs), enable the DBMS to reject operations that might corrupt the data. We present integrity constraints in the relational model in Section 3.2, along with a discussion of SQL support for ICs. We discuss how a DBMS enforces integrity constraints in Section 3.3. In Section 3.4 we turn to the mechanism for accessing and retrieving data from the database, **query languages**, and introduce the querying features of SQL, which we examine in greater detail in a later chapter.

We then discuss the step of converting an ER diagram into a relational database schema in Section 3.5. Finally, we introduce **views**, or tables defined using queries, in Section 3.6. Views can be used to define the external schema for a database and thus provide the support for logical data independence in the relational model.

### 3.1 INTRODUCTION TO THE RELATIONAL MODEL

The main construct for representing data in the relational model is a **relation**. A relation consists of a **relation schema** and a **relation instance**. The relation instance
is a table, and the relation schema describes the column heads for the table. We first describe the relation schema and then the relation instance. The schema specifies the relation’s name, the name of each field (or column, or attribute), and the domain of each field. A domain is referred to in a relation schema by the domain name and has a set of associated values.

We use the example of student information in a university database from Chapter 1 to illustrate the parts of a relation schema:

Students\((sid: \text{string, name: string, login: string, age: integer, gpa: real})\)

This says, for instance, that the field named \(sid\) has a domain named \text{string}. The set of values associated with domain \text{string} is the set of all character strings.

We now turn to the instances of a relation. An instance of a relation is a set of tuples, also called records, in which each tuple has the same number of fields as the relation schema. A relation instance can be thought of as a table in which each tuple is a row, and all rows have the same number of fields. (The term \text{relation instance} is often abbreviated to just \text{relation}, when there is no confusion with other aspects of a relation such as its schema.)

An instance of the Students relation appears in Figure 3.1. The instance \(S_1\) contains six tuples and has, as we expect from the schema, five fields. Note that no two rows are identical. This is a requirement of the relational model—each relation is defined to be a set of unique tuples or rows.\(^1\) The order in which the rows are listed is not important. Figure 3.2 shows the same relation instance. If the fields are named, as in

\(^1\)In practice, commercial systems allow tables to have duplicate rows, but we will assume that a relation is indeed a set of tuples unless otherwise noted.
Figure 3.2 An Alternative Representation of Instance S1 of Students

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Madayan</td>
<td>madayan@music</td>
<td>11</td>
<td>1.8</td>
</tr>
<tr>
<td>53832</td>
<td>Guldu</td>
<td>guldu@music</td>
<td>12</td>
<td>2.0</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>50000</td>
<td>Dave</td>
<td>dave@cs</td>
<td>19</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Our schema definitions and figures depicting relation instances, the order of fields does not matter either. However, an alternative convention is to list fields in a specific order and to refer to a field by its position. Thus sid is field 1 of Students, login is field 3, and so on. If this convention is used, the order of fields is significant. Most database systems use a combination of these conventions. For example, in SQL the named fields convention is used in statements that retrieve tuples, and the ordered fields convention is commonly used when inserting tuples.

A relation schema specifies the domain of each field or column in the relation instance. These **domain constraints** in the schema specify an important condition that we want each instance of the relation to satisfy: The values that appear in a column must be drawn from the domain associated with that column. Thus, the domain of a field is essentially the **type** of that field, in programming language terms, and restricts the values that can appear in the field.

More formally, let $R(f_1:D_1, \ldots, f_n:D_n)$ be a relation schema, and for each $f_i$, $1 \leq i \leq n$, let $Dom_i$ be the set of values associated with the domain named $D_i$. An instance of $R$ that satisfies the domain constraints in the schema is a set of tuples with $n$ fields:

$$\{ \langle f_1 : d_1, \ldots, f_n : d_n \rangle \mid d_1 \in Dom_1, \ldots, d_n \in Dom_n \}$$

The angular brackets $\langle \ldots \rangle$ identify the fields of a tuple. Using this notation, the first Students tuple shown in Figure 3.1 is written as $\langle sid: 50000, name: Dave, login: dave@cs, age: 19, gpa: 3.3 \rangle$. The curly brackets $\{ \ldots \}$ denote a set (of tuples, in this definition). The vertical bar $|$ should be read ‘such that,’ the symbol $\in$ should be read ‘in,’ and the expression to the right of the vertical bar is a condition that must be satisfied by the field values of each tuple in the set. Thus, an instance of $R$ is defined as a set of tuples. The fields of each tuple must correspond to the fields in the relation schema.

Domain constraints are so fundamental in the relational model that we will henceforth consider only relation instances that satisfy them; therefore, **relation instance** means **relation instance that satisfies the domain constraints in the relation schema**.
The degree, also called arity, of a relation is the number of fields. The cardinality of a relation instance is the number of tuples in it. In Figure 3.1, the degree of the relation (the number of columns) is five, and the cardinality of this instance is six.

A relational database is a collection of relations with distinct relation names. The relational database schema is the collection of schemas for the relations in the database. For example, in Chapter 1, we discussed a university database with relations called Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets_In. An instance of a relational database is a collection of relation instances, one per relation schema in the database schema; of course, each relation instance must satisfy the domain constraints in its schema.

3.1.1 Creating and Modifying Relations Using SQL-92

The SQL-92 language standard uses the word table to denote relation, and we will often follow this convention when discussing SQL. The subset of SQL that supports the creation, deletion, and modification of tables is called the Data Definition Language (DDL). Further, while there is a command that lets users define new domains, analogous to type definition commands in a programming language, we postpone a discussion of domain definition until Section 5.11. For now, we will just consider domains that are built-in types, such as integer.

The CREATE TABLE statement is used to define a new table. To create the Students relation, we can use the following statement:

```
CREATE TABLE Students ( sid CHAR(20),
                        name CHAR(30),
                        login CHAR(20),
                        age INTEGER,
                        gpa REAL )
```

Tuples are inserted using the INSERT command. We can insert a single tuple into the Students table as follows:

```
INSERT INTO Students (sid, name, login, age, gpa)
VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
```

We can optionally omit the list of column names in the INTO clause and list the values in the appropriate order, but it is good style to be explicit about column names.

---

2 SQL also provides statements to destroy tables and to change the columns associated with a table; we discuss these in Section 3.7.
We can delete tuples using the `DELETE` command. We can delete all Students tuples with `name` equal to Smith using the command:

```
DELETE
FROM Students S
WHERE S.name = 'Smith'
```

We can modify the column values in an existing row using the `UPDATE` command. For example, we can increment the age and decrement the gpa of the student with `sid` 53688:

```
UPDATE Students S
SET S.age = S.age + 1, S.gpa = S.gpa - 1
WHERE S.sid = 53688
```

These examples illustrate some important points. The `WHERE` clause is applied first and determines which rows are to be modified. The `SET` clause then determines how these rows are to be modified. If the column that is being modified is also used to determine the new value, the value used in the expression on the right side of equals (=) is the old value, that is, before the modification. To illustrate these points further, consider the following variation of the previous query:

```
UPDATE Students S
SET S.gpa = S.gpa - 0.1
WHERE S.gpa >= 3.3
```

If this query is applied on the instance `S1` of Students shown in Figure 3.1, we obtain the instance shown in Figure 3.3.

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
<td>Dave</td>
<td>dave@cs</td>
<td>19</td>
<td>3.2</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.3</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.7</td>
</tr>
<tr>
<td>53831</td>
<td>Madayan</td>
<td>madayan@music</td>
<td>11</td>
<td>1.8</td>
</tr>
<tr>
<td>53832</td>
<td>Guldu</td>
<td>guldu@music</td>
<td>12</td>
<td>2.0</td>
</tr>
</tbody>
</table>

**Figure 3.3** Students Instance `S1` after Update

### 3.2 INTEGRITY CONSTRAINTS OVER RELATIONS

A database is only as good as the information stored in it, and a DBMS must therefore help prevent the entry of incorrect information. An integrity constraint (IC) is a
A condition that is specified on a database schema, and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constraints specified on the database schema, it is a **legal** instance. A DBMS enforces integrity constraints, in that it permits only legal instances to be stored in the database.

Integrity constraints are specified and enforced at different times:

1. When the DBA or end user defines a database schema, he or she specifies the ICs that must hold on any instance of this database.

2. When a database application is run, the DBMS checks for violations and disallows changes to the data that violate the specified ICs. (In some situations, rather than disallow the change, the DBMS might instead make some compensating changes to the data to ensure that the database instance satisfies all ICs. In any case, changes to the database are not allowed to create an instance that violates any IC.)

Many kinds of integrity constraints can be specified in the relational model. We have already seen one example of an integrity constraint in the *domain constraints* associated with a relation schema (Section 3.1). In general, other kinds of constraints can be specified as well; for example, no two students have the same *sid* value. In this section we discuss the integrity constraints, other than domain constraints, that a DBA or user can specify in the relational model.

### 3.2.1 Key Constraints

Consider the Students relation and the constraint that no two students have the same student id. This IC is an example of a key constraint. A **key constraint** is a statement that a certain *minimal* subset of the fields of a relation is a unique identifier for a tuple. A set of fields that uniquely identifies a tuple according to a key constraint is called a **candidate key** for the relation; we often abbreviate this to just *key*. In the case of the Students relation, the (set of fields containing just the) *sid* field is a candidate key.

Let us take a closer look at the above definition of a (candidate) key. There are two parts to the definition:

1. Two distinct tuples in a legal instance (an instance that satisfies all ICs, including the key constraint) cannot have identical values in all the fields of a key.

2. No subset of the set of fields in a key is a unique identifier for a tuple.

---

3The term *key* is rather overworked. In the context of access methods, we speak of *search keys*, which are quite different.
Chapter 3

The first part of the definition means that in any legal instance, the values in the key fields uniquely identify a tuple in the instance. When specifying a key constraint, the DBA or user must be sure that this constraint will not prevent them from storing a ‘correct’ set of tuples. (A similar comment applies to the specification of other kinds of ICs as well.) The notion of ‘correctness’ here depends upon the nature of the data being stored. For example, several students may have the same name, although each student has a unique student id. If the name field is declared to be a key, the DBMS will not allow the Students relation to contain two tuples describing different students with the same name!

The second part of the definition means, for example, that the set of fields \{sid, name\} is not a key for Students, because this set properly contains the key \{sid\}. The set \{sid, name\} is an example of a superkey, which is a set of fields that contains a key.

Look again at the instance of the Students relation in Figure 3.1. Observe that two different rows always have different sid values; sid is a key and uniquely identifies a tuple. However, this does not hold for nonkey fields. For example, the relation contains two rows with Smith in the name field.

Note that every relation is guaranteed to have a key. Since a relation is a set of tuples, the set of all fields is always a superkey. If other constraints hold, some subset of the fields may form a key, but if not, the set of all fields is a key.

A relation may have several candidate keys. For example, the login and age fields of the Students relation may, taken together, also identify students uniquely. That is, \{login, age\} is also a key. It may seem that login is a key, since no two rows in the example instance have the same login value. However, the key must identify tuples uniquely in all possible legal instances of the relation. By stating that \{login, age\} is a key, the user is declaring that two students may have the same login or age, but not both.

Out of all the available candidate keys, a database designer can identify a primary key. Intuitively, a tuple can be referred to from elsewhere in the database by storing the values of its primary key fields. For example, we can refer to a Students tuple by storing its sid value. As a consequence of referring to student tuples in this manner, tuples are frequently accessed by specifying their sid value. In principle, we can use any key, not just the primary key, to refer to a tuple. However, using the primary key is preferable because it is what the DBMS expects—this is the significance of designating a particular candidate key as a primary key—and optimizes for. For example, the DBMS may create an index with the primary key fields as the search key, to make the retrieval of a tuple given its primary key value efficient. The idea of referring to a tuple is developed further in the next section.
Specifying Key Constraints in SQL-92

In SQL we can declare that a subset of the columns of a table constitute a key by using the \texttt{UNIQUE} constraint. At most one of these ‘candidate’ keys can be declared to be a \textit{primary key}, using the \texttt{PRIMARY KEY} constraint. (SQL does not require that such constraints be declared for a table.)

Let us revisit our example table definition and specify key information:

\begin{verbatim}
CREATE TABLE Students ( sid CHAR(20),
                 name CHAR(30),
                 login CHAR(20),
                 age INTEGER,
                 gpa REAL,
                 UNIQUE (name, age),
                 CONSTRAINT StudentsKey PRIMARY KEY (sid) )
\end{verbatim}

This definition says that \texttt{sid} is the primary key and that the combination of \texttt{name} and \texttt{age} is also a key. The definition of the primary key also illustrates how we can name a constraint by preceding it with \texttt{CONSTRAINT constraint-name}. If the constraint is violated, the constraint name is returned and can be used to identify the error.

3.2.2 Foreign Key Constraints

Sometimes the information stored in a relation is linked to the information stored in another relation. If one of the relations is modified, the other must be checked, and perhaps modified, to keep the data consistent. An IC involving both relations must be specified if a DBMS is to make such checks. The most common IC involving two relations is a \textit{foreign key} constraint.

Suppose that in addition to Students, we have a second relation:

\begin{verbatim}
Enrolled(sid: string, cid: string, grade: string)
\end{verbatim}

To ensure that only bona fide students can enroll in courses, any value that appears in the \texttt{sid} field of an instance of the Enrolled relation should also appear in the \texttt{sid} field of some tuple in the Students relation. The \texttt{sid} field of Enrolled is called a \textit{foreign key} and \textit{refers} to Students. The foreign key in the referencing relation (Enrolled, in our example) must match the primary key of the referenced relation (Students), i.e., it must have the same number of columns and compatible data types, although the column names can be different.

This constraint is illustrated in Figure 3.4. As the figure shows, there may well be some students who are not referenced from Enrolled (e.g., the student with \texttt{sid=50000}).
However, every sid value that appears in the instance of the Enrolled table appears in the primary key column of a row in the Students table.

If we try to insert the tuple \( \langle 55555, \text{Art104}, A \rangle \) into \( E_1 \), the IC is violated because there is no tuple in \( S_1 \) with the id 55555; the database system should reject such an insertion. Similarly, if we delete the tuple \( \langle 53666, \text{Jones}, jones@cs, 18, 3.4 \rangle \) from \( S_1 \), we violate the foreign key constraint because the tuple \( \langle 53666, \text{History105}, B \rangle \) in \( E_1 \) contains sid value 53666, the sid of the deleted Students tuple. The DBMS should disallow the deletion or, perhaps, also delete the Enrolled tuple that refers to the deleted Students tuple. We discuss foreign key constraints and their impact on updates in Section 3.3.

Finally, we note that a foreign key could refer to the same relation. For example, we could extend the Students relation with a column called \textit{partner} and declare this column to be a foreign key referring to Students. Intuitively, every student could then have a partner, and the \textit{partner} field contains the partner’s sid. The observant reader will no doubt ask, “What if a student does not (yet) have a partner?” This situation is handled in SQL by using a special value called \textit{null}. The use of \textit{null} in a field of a tuple means that value in that field is either unknown or not applicable (e.g., we do not know the partner yet, or there is no partner). The appearance of \textit{null} in a foreign key field does not violate the foreign key constraint. However, \textit{null} values are not allowed to appear in a primary key field (because the primary key fields are used to identify a tuple uniquely). We will discuss \textit{null} values further in Chapter 5.

**Specifying Foreign Key Constraints in SQL-92**

Let us define Enrolled(\textit{sid}: string, \textit{cid}: string, \textit{grade}: string):

\[
\text{CREATE TABLE Enrolled ( sid CHAR(20), }
\]

Figure 3.4 Referential Integrity
The foreign key constraint states that every \textit{sid} value in Enrolled must also appear in Students, that is, \textit{sid} in Enrolled is a foreign key referencing Students. Incidentally, the primary key constraint states that a student has exactly one grade for each course that he or she is enrolled in. If we want to record more than one grade per student per course, we should change the primary key constraint.

### 3.2.3 General Constraints

Domain, primary key, and foreign key constraints are considered to be a fundamental part of the relational data model and are given special attention in most commercial systems. Sometimes, however, it is necessary to specify more general constraints.

For example, we may require that student ages be within a certain range of values; given such an IC specification, the DBMS will reject inserts and updates that violate the constraint. This is very useful in preventing data entry errors. If we specify that all students must be at least 16 years old, the instance of Students shown in Figure 3.1 is illegal because two students are underage. If we disallow the insertion of these two tuples, we have a legal instance, as shown in Figure 3.5.

![Figure 3.5](image)

The IC that students must be older than 16 can be thought of as an extended domain constraint, since we are essentially defining the set of permissible \textit{age} values more stringently than is possible by simply using a standard domain such as \textit{integer}. In general, however, constraints that go well beyond domain, key, or foreign key constraints can be specified. For example, we could require that every student whose age is greater than 18 must have a \textit{gpa} greater than 3.

Current relational database systems support such general constraints in the form of \textit{table constraints} and \textit{assertions}. Table constraints are associated with a single table and are checked whenever that table is modified. In contrast, assertions involve several
tables and are checked whenever any of these tables is modified. Both table constraints and assertions can use the full power of SQL queries to specify the desired restriction. We discuss SQL support for table constraints and assertions in Section 5.11 because a full appreciation of their power requires a good grasp of SQL’s query capabilities.

3.3 ENFORCING INTEGRITY CONSTRAINTS

As we observed earlier, ICs are specified when a relation is created and enforced when a relation is modified. The impact of domain, PRIMARY KEY, and UNIQUE constraints is straightforward: if an insert, delete, or update command causes a violation, it is rejected. Potential IC violation is generally checked at the end of each SQL statement execution, although it can be deferred until the end of the transaction executing the statement, as we will see in Chapter 18.

Consider the instance $S_1$ of Students shown in Figure 3.1. The following insertion violates the primary key constraint because there is already a tuple with the sid 53688, and it will be rejected by the DBMS:

```sql
INSERT INTO Students (sid, name, login, age, gpa)
VALUES (53688, 'Mike', 'mike@ee', 17, 3.4)
```

The following insertion violates the constraint that the primary key cannot contain null:

```sql
INSERT INTO Students (sid, name, login, age, gpa)
VALUES (null, 'Mike', 'mike@ee', 17, 3.4)
```

Of course, a similar problem arises whenever we try to insert a tuple with a value in a field that is not in the domain associated with that field, i.e., whenever we violate a domain constraint. Deletion does not cause a violation of domain, primary key or unique constraints. However, an update can cause violations, similar to an insertion:

```sql
UPDATE Students S
SET S.sid = 50000
WHERE S.sid = 53688
```

This update violates the primary key constraint because there is already a tuple with sid 50000.

The impact of foreign key constraints is more complex because SQL sometimes tries to rectify a foreign key constraint violation instead of simply rejecting the change. We will
discuss the referential integrity enforcement steps taken by the DBMS in terms of our Enrolled and Students tables, with the foreign key constraint that Enrolled.sid is a reference to (the primary key of) Students.

In addition to the instance S1 of Students, consider the instance of Enrolled shown in Figure 3.4. Deletions of Enrolled tuples do not violate referential integrity, but insertions of Enrolled tuples could. The following insertion is illegal because there is no student with sid 51111:

```sql
INSERT INTO Enrolled (cid, grade, sid)
VALUES ('Hindi101', 'B', 51111)
```

On the other hand, insertions of Students tuples do not violate referential integrity although deletions could. Further, updates on either Enrolled or Students that change the sid value could potentially violate referential integrity.

SQL-92 provides several alternative ways to handle foreign key violations. We must consider three basic questions:

1. **What should we do if an Enrolled row is inserted, with a sid column value that does not appear in any row of the Students table?**

   In this case the INSERT command is simply rejected.

2. **What should we do if a Students row is deleted?**

   The options are:
   - Delete all Enrolled rows that refer to the deleted Students row.
   - Disallow the deletion of the Students row if an Enrolled row refers to it.
   - Set the sid column to the sid of some (existing) ‘default’ student, for every Enrolled row that refers to the deleted Students row.
   - For every Enrolled row that refers to it, set the sid column to null. In our example, this option conflicts with the fact that sid is part of the primary key of Enrolled and therefore cannot be set to null. Thus, we are limited to the first three options in our example, although this fourth option (setting the foreign key to null) is available in the general case.

3. **What should we do if the primary key value of a Students row is updated?**

   The options here are similar to the previous case.

SQL-92 allows us to choose any of the four options on DELETE and UPDATE. For example, we can specify that when a Students row is deleted, all Enrolled rows that refer to it are to be deleted as well, but that when the sid column of a Students row is modified, this update is to be rejected if an Enrolled row refers to the modified Students row:
CREATE TABLE Enrolled ( sid CHAR(20),
cid CHAR(20),
grade CHAR(10),
PRIMARY KEY (sid, cid),
FOREIGN KEY (sid) REFERENCES Students
ON DELETE CASCADE
ON UPDATE NO ACTION )

The options are specified as part of the foreign key declaration. The default option is NO ACTION, which means that the action (DELETE or UPDATE) is to be rejected. Thus, the ON UPDATE clause in our example could be omitted, with the same effect. The CASCADE keyword says that if a Students row is deleted, all Enrolled rows that refer to it are to be deleted as well. If the UPDATE clause specified CASCADE, and the sid column of a Students row is updated, this update is also carried out in each Enrolled row that refers to the updated Students row.

If a Students row is deleted, we can switch the enrollment to a ‘default’ student by using ON DELETE SET DEFAULT. The default student is specified as part of the definition of the sid field in Enrolled; for example, sid CHAR(20) DEFAULT '53666'. Although the specification of a default value is appropriate in some situations (e.g., a default parts supplier if a particular supplier goes out of business), it is really not appropriate to switch enrollments to a default student. The correct solution in this example is to also delete all enrollment tuples for the deleted student (that is, CASCADE), or to reject the update.

SQL also allows the use of null as the default value by specifying ON DELETE SET NULL.

### 3.4 QUERYING RELATIONAL DATA

A relational database query (query, for short) is a question about the data, and the answer consists of a new relation containing the result. For example, we might want to find all students younger than 18 or all students enrolled in Reggae203. A query language is a specialized language for writing queries.

SQL is the most popular commercial query language for a relational DBMS. We now present some SQL examples that illustrate how easily relations can be queried. Consider the instance of the Students relation shown in Figure 3.1. We can retrieve rows corresponding to students who are younger than 18 with the following SQL query:

```
SELECT *
FROM Students S
WHERE S.age < 18
```
The symbol * means that we retain all fields of selected tuples in the result. To understand this query, think of S as a variable that takes on the value of each tuple in Students, one tuple after the other. The condition $S.age < 18$ in the WHERE clause specifies that we want to select only tuples in which the age field has a value less than 18. This query evaluates to the relation shown in Figure 3.6.

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Madayan</td>
<td>madayan@music</td>
<td>11</td>
<td>1.8</td>
</tr>
<tr>
<td>53832</td>
<td>Guldu</td>
<td>guldu@music</td>
<td>12</td>
<td>2.0</td>
</tr>
</tbody>
</table>

**Figure 3.6** Students with age < 18 on Instance S1

This example illustrates that the domain of a field restricts the operations that are permitted on field values, in addition to restricting the values that can appear in the field. The condition $S.age < 18$ involves an arithmetic comparison of an age value with an integer and is permissible because the domain of age is the set of integers. On the other hand, a condition such as $S.age = S.sid$ does not make sense because it compares an integer value with a string value, and this comparison is defined to fail in SQL; a query containing this condition will produce no answer tuples.

In addition to selecting a subset of tuples, a query can extract a subset of the fields of each selected tuple. We can compute the names and logins of students who are younger than 18 with the following query:

```sql
SELECT S.name, S.login
FROM Students S
WHERE S.age < 18
```

Figure 3.7 shows the answer to this query; it is obtained by applying the selection to the instance S1 of Students (to get the relation shown in Figure 3.6), followed by removing unwanted fields. Note that the order in which we perform these operations does matter—if we remove unwanted fields first, we cannot check the condition $S.age < 18$, which involves one of those fields.

We can also combine information in the Students and Enrolled relations. If we want to obtain the names of all students who obtained an A and the id of the course in which they got an A, we could write the following query:

```sql
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid = E.sid AND E.grade = 'A'
```
**DISTINCT types in SQL:** A comparison of two values drawn from different domains should fail, even if the values are ‘compatible’ in the sense that both are numeric or both are string values etc. For example, if salary and age are two different domains whose values are represented as integers, a comparison of a salary value with an age value should fail. Unfortunately, SQL-92’s support for the concept of domains does not go this far: We are forced to define salary and age as integer types and the comparison $S < A$ will succeed when $S$ is bound to the salary value 25 and $A$ is bound to the age value 50. The latest version of the SQL standard, called SQL:1999, addresses this problem, and allows us to define salary and age as **DISTINCT** types even though their values are represented as integers. Many systems, e.g., Informix UDS and IBM DB2, already support this feature.

<table>
<thead>
<tr>
<th>name</th>
<th>login</th>
</tr>
</thead>
<tbody>
<tr>
<td>Madayan</td>
<td>madayan@music</td>
</tr>
<tr>
<td>Guldu</td>
<td>guldu@music</td>
</tr>
</tbody>
</table>

**Figure 3.7** Names and Logins of Students under 18

This query can be understood as follows: “If there is a Students tuple $S$ and an Enrolled tuple $E$ such that $S.sid = E.sid$ (so that $S$ describes the student who is enrolled in $E$) and $E.grade = 'A'$, then print the student’s name and the course id.” When evaluated on the instances of Students and Enrolled in Figure 3.4, this query returns a single tuple, $⟨Smith, Topology112⟩$.

We will cover relational queries, and SQL in particular, in more detail in subsequent chapters.

### 3.5 LOGICAL DATABASE DESIGN: ER TO RELATIONAL

The ER model is convenient for representing an initial, high-level database design. Given an ER diagram describing a database, there is a standard approach to generating a relational database schema that closely approximates the ER design. (The translation is approximate to the extent that we cannot capture all the constraints implicit in the ER design using SQL-92, unless we use certain SQL-92 constraints that are costly to check.) We now describe how to translate an ER diagram into a collection of tables with associated constraints, i.e., a relational database schema.
3.5.1 Entity Sets to Tables

An entity set is mapped to a relation in a straightforward way: Each attribute of the entity set becomes an attribute of the table. Note that we know both the domain of each attribute and the (primary) key of an entity set.

Consider the Employees entity set with attributes ssn, name, and lot shown in Figure 3.8. A possible instance of the Employees entity set, containing three Employees entities, is shown in Figure 3.9 in a tabular format.

<table>
<thead>
<tr>
<th>ssn</th>
<th>name</th>
<th>lot</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-22-3666</td>
<td>Attishoo</td>
<td>48</td>
</tr>
<tr>
<td>231-31-5368</td>
<td>Smiley</td>
<td>22</td>
</tr>
<tr>
<td>131-24-3650</td>
<td>Smethurst</td>
<td>35</td>
</tr>
</tbody>
</table>

The following SQL statement captures the preceding information, including the domain constraints and key information:

```sql
CREATE TABLE Employees ( ssn CHAR(11),
                         name CHAR(30),
                         lot INTEGER,
                         PRIMARY KEY (ssn) )
```

3.5.2 Relationship Sets (without Constraints) to Tables

A relationship set, like an entity set, is mapped to a relation in the relational model. We begin by considering relationship sets without key and participation constraints, and we discuss how to handle such constraints in subsequent sections. To represent a relationship, we must be able to identify each participating entity and give values
to the descriptive attributes of the relationship. Thus, the attributes of the relation include:

- The primary key attributes of each participating entity set, as foreign key fields.
- The descriptive attributes of the relationship set.

The set of nondescriptive attributes is a superkey for the relation. If there are no key constraints (see Section 2.4.1), this set of attributes is a candidate key.

Consider the Works\_In2 relationship set shown in Figure 3.10. Each department has offices in several locations and we want to record the locations at which each employee works.

All the available information about the Works\_In2 table is captured by the following SQL definition:

```
CREATE TABLE Works\_In2 ( ssn CHAR(11),
  did INTEGER,
  address CHAR(20),
  since DATE,
  PRIMARY KEY (ssn, did, address),
  FOREIGN KEY (ssn) REFERENCES Employees,
  FOREIGN KEY (address) REFERENCES Locations,
  FOREIGN KEY (did) REFERENCES Departments )
```

Note that the address, did, and ssn fields cannot take on null values. Because these fields are part of the primary key for Works\_In2, a NOT NULL constraint is implicit for each of these fields. This constraint ensures that these fields uniquely identify a department, an employee, and a location in each tuple of Works\_In. We can also
specify that a particular action is desired when a referenced Employees, Departments or Locations tuple is deleted, as explained in the discussion of integrity constraints in Section 3.2. In this chapter we assume that the default action is appropriate except for situations in which the semantics of the ER diagram require some other action.

Finally, consider the Reports-To relationship set shown in Figure 3.11. The role indicators supervisor and subordinate are used to create meaningful field names in the CREATE statement for the Reports-To table:

```
CREATE TABLE Reports_To (  
  supervisor_ssn  CHAR(11),
  subordinate_ssn  CHAR(11),
  PRIMARY KEY (supervisor_ssn, subordinate_ssn),
  FOREIGN KEY (supervisor_ssn) REFERENCES Employees(ssn),
  FOREIGN KEY (subordinate_ssn) REFERENCES Employees(ssn) )
```

Observe that we need to explicitly name the referenced field of Employees because the field name differs from the name(s) of the referring field(s).

### 3.5.3 Translating Relationship Sets with Key Constraints

If a relationship set involves \( n \) entity sets and some \( m \) of them are linked via arrows in the ER diagram, the key for any one of these \( m \) entity sets constitutes a key for the relation to which the relationship set is mapped. Thus we have \( m \) candidate keys, and one of these should be designated as the primary key. The translation discussed in Section 2.3 from relationship sets to a relation can be used in the presence of key constraints, taking into account this point about keys.
Consider the relationship set Manages shown in Figure 3.12. The table corresponding to Manages has the attributes \textit{ssn}, \textit{did}, \textit{since}. However, because each department has at most one manager, no two tuples can have the same \textit{did} value but differ on the \textit{ssn} value. A consequence of this observation is that \textit{did} is itself a key for Manages; indeed, the set \textit{did}, \textit{ssn} is not a key (because it is not minimal). The Manages relation can be defined using the following SQL statement:

```sql
CREATE TABLE Manages ( ssn CHAR(11), did INTEGER, since DATE, PRIMARY KEY (did), FOREIGN KEY (ssn) REFERENCES Employees, FOREIGN KEY (did) REFERENCES Departments )
```

A second approach to translating a relationship set with key constraints is often superior because it avoids creating a distinct table for the relationship set. The idea is to include the information about the relationship set in the table corresponding to the entity set with the key, taking advantage of the key constraint. In the Manages example, because a department has at most one manager, we can add the key fields of the Employees tuple denoting the manager and the \textit{since} attribute to the Departments tuple.

This approach eliminates the need for a separate Manages relation, and queries asking for a department’s manager can be answered without combining information from two relations. The only drawback to this approach is that space could be wasted if several departments have no managers. In this case the added fields would have to be filled with \textit{null} values. The first translation (using a separate table for Manages) avoids this inefficiency, but some important queries require us to combine information from two relations, which can be a slow operation.

The following SQL statement, defining a Dept.Mgr relation that captures the information in both Departments and Manages, illustrates the second approach to translating relationship sets with key constraints:
The Relational Model

CREATE TABLE Dept_Mgr ( did INTEGER,
    dname CHAR(20),
    budget REAL,
    ssn CHAR(11),
    since DATE,
    PRIMARY KEY (did),
    FOREIGN KEY (ssn) REFERENCES Employees )

Note that ssn can take on null values.

This idea can be extended to deal with relationship sets involving more than two entity sets. In general, if a relationship set involves \( n \) entity sets and some \( m \) of them are linked via arrows in the ER diagram, the relation corresponding to any one of the \( m \) sets can be augmented to capture the relationship.

We discuss the relative merits of the two translation approaches further after considering how to translate relationship sets with participation constraints into tables.

3.5.4 Translating Relationship Sets with Participation Constraints

Consider the ER diagram in Figure 3.13, which shows two relationship sets, Manages and Works_In.

![Figure 3.13 Manages and Works_In](image-url)
Every department is required to have a manager, due to the participation constraint, and at most one manager, due to the key constraint. The following SQL statement reflects the second translation approach discussed in Section 3.5.3, and uses the key constraint:

```
CREATE TABLE Dept_Mgr ( did INTEGER, 
dname CHAR(20), 
budget REAL, 
ssn CHAR(11) NOT NULL, 
since DATE, 
PRIMARY KEY (did), 
FOREIGN KEY (ssn) REFERENCES Employees 
ON DELETE NO ACTION )
```

It also captures the participation constraint that every department must have a manager: Because `ssn` cannot take on `null` values, each tuple of Dept_Mgr identifies a tuple in Employees (who is the manager). The `NO ACTION` specification, which is the default and need not be explicitly specified, ensures that an Employees tuple cannot be deleted while it is pointed to by a Dept_Mgr tuple. If we wish to delete such an Employees tuple, we must first change the Dept_Mgr tuple to have a new employee as manager. (We could have specified `CASCADE` instead of `NO ACTION`, but deleting all information about a department just because its manager has been fired seems a bit extreme!)

The constraint that every department must have a manager cannot be captured using the first translation approach discussed in Section 3.5.3. (Look at the definition of Manages and think about what effect it would have if we added `NOT NULL` constraints to the `ssn` and `did` fields. Hint: The constraint would prevent the firing of a manager, but does not ensure that a manager is initially appointed for each department!) This situation is a strong argument in favor of using the second approach for one-to-many relationships such as Manages, especially when the entity set with the key constraint also has a total participation constraint.

Unfortunately, there are many participation constraints that we cannot capture using SQL-92, short of using `table constraints` or `assertions`. Table constraints and assertions can be specified using the full power of the SQL query language (as discussed in Section 5.11) and are very expressive, but also very expensive to check and enforce. For example, we cannot enforce the participation constraints on the Works_In relation without using these general constraints. To see why, consider the Works_In relation obtained by translating the ER diagram into relations. It contains fields `ssn` and `did`, which are foreign keys referring to Employees and Departments. To ensure total participation of Departments in Works_In, we have to guarantee that every `did` value in Departments appears in a tuple of Works_In. We could try to guarantee this condition by declaring that `did` in Departments is a foreign key referring to Works_In, but this is not a valid foreign key constraint because `did` is not a candidate key for Works_In.
To ensure total participation of Departments in Works_In using SQL-92, we need an assertion. We have to guarantee that every did value in Departments appears in a tuple of Works_In; further, this tuple of Works_In must also have non-null values in the fields that are foreign keys referencing other entity sets involved in the relationship (in this example, the ssn field). We can ensure the second part of this constraint by imposing the stronger requirement that ssn in Works_In cannot contain null values. (Ensuring that the participation of Employees in Works_In is total is symmetric.)

Another constraint that requires assertions to express in SQL is the requirement that each Employees entity (in the context of the Manages relationship set) must manage at least one department.

In fact, the Manages relationship set exemplifies most of the participation constraints that we can capture using key and foreign key constraints. Manages is a binary relationship set in which exactly one of the entity sets (Departments) has a key constraint, and the total participation constraint is expressed on that entity set.

We can also capture participation constraints using key and foreign key constraints in one other special situation: a relationship set in which all participating entity sets have key constraints and total participation. The best translation approach in this case is to map all the entities as well as the relationship into a single table; the details are straightforward.

### 3.5.5 Translating Weak Entity Sets

A weak entity set always participates in a one-to-many binary relationship and has a key constraint and total participation. The second translation approach discussed in Section 3.5.3 is ideal in this case, but we must take into account the fact that the weak entity has only a partial key. Also, when an owner entity is deleted, we want all owned weak entities to be deleted.

Consider the Dependents weak entity set shown in Figure 3.14, with partial key pname. A Dependents entity can be identified uniquely only if we take the key of the owning Employees entity and the pname of the Dependents entity, and the Dependents entity must be deleted if the owning Employees entity is deleted.

We can capture the desired semantics with the following definition of the Dep_Policy relation:

```sql
CREATE TABLE Dep_Policy ( pname CHAR(20), age INTEGER, cost REAL, ssn CHAR(11),
```
Observe that the primary key is \( \langle \text{pname}, \text{ssn} \rangle \), since Dependents is a weak entity. This constraint is a change with respect to the translation discussed in Section 3.5.3. We have to ensure that every Dependents entity is associated with an Employees entity (the owner), as per the total participation constraint on Dependents. That is, \( \text{ssn} \) cannot be null. This is ensured because \( \text{ssn} \) is part of the primary key. The \text{CASCADE} option ensures that information about an employee's policy and dependents is deleted if the corresponding Employees tuple is deleted.

### 3.5.6 Translating Class Hierarchies

We present the two basic approaches to handling ISA hierarchies by applying them to the ER diagram shown in Figure 3.15:
1. We can map each of the entity sets Employees, Hourly_Emps, and Contract_Emps to a distinct relation. The Employees relation is created as in Section 2.2. We discuss Hourly_Emps here; Contract_Emps is handled similarly. The relation for Hourly_Emps includes the *hourly_wages* and *hours_worked* attributes of Hourly_Emps. It also contains the key attributes of the superclass (*ssn*, in this example), which serve as the primary key for Hourly_Emps, as well as a foreign key referencing the superclass (Employees). For each Hourly_Emps entity, the value of the *name* and *lot* attributes are stored in the corresponding row of the superclass (Employees). Note that if the superclass tuple is deleted, the delete must be cascaded to Hourly_Emps.

2. Alternatively, we can create just two relations, corresponding to Hourly_Emps and Contract_Emps. The relation for Hourly_Emps includes all the attributes of Hourly_Emps as well as all the attributes of Employees (i.e., *ssn*, *name*, *lot*, *hourly_wages*, *hours_worked*).

The first approach is general and is always applicable. Queries in which we want to examine all employees and do not care about the attributes specific to the subclasses are handled easily using the Employees relation. However, queries in which we want to examine, say, hourly employees, may require us to combine Hourly_Emps (or Contract_Emps, as the case may be) with Employees to retrieve *name* and *lot*.

The second approach is not applicable if we have employees who are neither hourly employees nor contract employees, since there is no way to store such employees. Also, if an employee is both an Hourly_Emps and a Contract_Emps entity, then the *name* and *lot* values are stored twice. This duplication can lead to some of the anomalies that we discuss in Chapter 15. A query that needs to examine all employees must now examine two relations. On the other hand, a query that needs to examine only hourly employees can now do so by examining just one relation. The choice between these approaches clearly depends on the semantics of the data and the frequency of common operations.

In general, overlap and covering constraints can be expressed in SQL-92 only by using assertions.

### 3.5.7 Translating ER Diagrams with Aggregation

Translating aggregation into the relational model is easy because there is no real distinction between entities and relationships in the relational model.

Consider the ER diagram shown in Figure 3.16. The Employees, Projects, and Departments entity sets and the Sponsors relationship set are mapped as described in previous sections. For the Monitors relationship set, we create a relation with the following attributes: the key attributes of Employees (*ssn*), the key attributes of Spon-
sors \((\text{did}, \text{pid})\), and the descriptive attributes of Monitors \((\text{until})\). This translation is essentially the standard mapping for a relationship set, as described in Section 3.5.2.

There is a special case in which this translation can be refined further by dropping the Sponsors relation. Consider the Sponsors relation. It has attributes \(\text{pid}, \text{did}, \text{and since}\), and in general we need it (in addition to Monitors) for two reasons:

1. We have to record the descriptive attributes (in our example, \(\text{since}\)) of the Sponsors relationship.
2. Not every sponsorship has a monitor, and thus some \(\langle \text{pid}, \text{did} \rangle\) pairs in the Sponsors relation may not appear in the Monitors relation.

However, if Sponsors has no descriptive attributes and has total participation in Monitors, every possible instance of the Sponsors relation can be obtained by looking at the \(\langle \text{pid}, \text{did} \rangle\) columns of the Monitors relation. Thus, we need not store the Sponsors relation in this case.

### 3.5.8 ER to Relational: Additional Examples *

Consider the ER diagram shown in Figure 3.17. We can translate this ER diagram into the relational model as follows, taking advantage of the key constraints to combine Purchaser information with Policies and Beneficiary information with Dependents:
CREATE TABLE Policies ( policyid INTEGER, 
cost REAL, 
ssn CHAR(11) NOT NULL, 
PRIMARY KEY (policyid), 
FOREIGN KEY (ssn) REFERENCES Employees 
ON DELETE CASCADE )

CREATE TABLE Dependents ( pname CHAR(20), 
age INTEGER, 
policyid INTEGER, 
PRIMARY KEY (pname, policyid), 
FOREIGN KEY (policyid) REFERENCES Policies 
ON DELETE CASCADE )

Notice how the deletion of an employee leads to the deletion of all policies owned by the employee and all dependents who are beneficiaries of those policies. Further, each dependent is required to have a covering policy—because policyid is part of the primary key of Dependents, there is an implicit NOT NULL constraint. This model accurately reflects the participation constraints in the ER diagram and the intended actions when an employee entity is deleted.

In general, there could be a chain of identifying relationships for weak entity sets. For example, we assumed that policyid uniquely identifies a policy. Suppose that policyid only distinguishes the policies owned by a given employee; that is, policyid is only a partial key and Policies should be modeled as a weak entity set. This new assumption
about policyid does not cause much to change in the preceding discussion. In fact, the only changes are that the primary key of Policies becomes \( \langle \text{policyid}, \text{ssn} \rangle \), and as a consequence, the definition of Dependents changes—a field called ssn is added and becomes part of both the primary key of Dependents and the foreign key referencing Policies:

```sql
CREATE TABLE Dependents (  pname CHAR(20),  ssn CHAR(11),  age INTEGER,  policyid INTEGER NOT NULL,  PRIMARY KEY (pname, policyid, ssn),  FOREIGN KEY (policyid, ssn) REFERENCES Policies  ON DELETE CASCADE)
```

### 3.6 INTRODUCTION TO VIEWS

A view is a table whose rows are not explicitly stored in the database but are computed as needed from a view definition. Consider the Students and Enrolled relations. Suppose that we are often interested in finding the names and student identifiers of students who got a grade of B in some course, together with the cid for the course. We can define a view for this purpose. Using SQL-92 notation:

```sql
CREATE VIEW B-Students (name, sid, course)  AS SELECT S.sname, S.sid, E.cid  FROM Students S, Enrolled E  WHERE S.sid = E.sid AND E.grade = 'B'
```

The view B-Students has three fields called name, sid, and course with the same domains as the fields sname and sid in Students and cid in Enrolled. (If the optional arguments name, sid, and course are omitted from the CREATE VIEW statement, the column names sname, sid, and cid are inherited.)

This view can be used just like a base table, or explicitly stored table, in defining new queries or views. Given the instances of Enrolled and Students shown in Figure 3.4, B-Students contains the tuples shown in Figure 3.18. Conceptually, whenever B-Students is used in a query, the view definition is first evaluated to obtain the corresponding instance of B-Students, and then the rest of the query is evaluated treating B-Students like any other relation referred to in the query. (We will discuss how queries on views are evaluated in practice in Chapter 23.)
3.6.1 Views, Data Independence, Security

Consider the levels of abstraction that we discussed in Section 1.5.2. The physical schema for a relational database describes how the relations in the conceptual schema are stored, in terms of the file organizations and indexes used. The conceptual schema is the collection of schemas of the relations stored in the database. While some relations in the conceptual schema can also be exposed to applications, i.e., be part of the external schema of the database, additional relations in the external schema can be defined using the view mechanism. The view mechanism thus provides the support for logical data independence in the relational model. That is, it can be used to define relations in the external schema that mask changes in the conceptual schema of the database from applications. For example, if the schema of a stored relation is changed, we can define a view with the old schema, and applications that expect to see the old schema can now use this view.

Views are also valuable in the context of security. We can define views that give a group of users access to just the information they are allowed to see. For example, we can define a view that allows students to see other students’ name and age but not their gpa, and allow all students to access this view, but not the underlying Students table (see Chapter 17).

3.6.2 Updates on Views

The motivation behind the view mechanism is to tailor how users see the data. Users should not have to worry about the view versus base table distinction. This goal is indeed achieved in the case of queries on views; a view can be used just like any other relation in defining a query. However, it is natural to want to specify updates on views as well. Here, unfortunately, the distinction between a view and a base table must be kept in mind.

The SQL-92 standard allows updates to be specified only on views that are defined on a single base table using just selection and projection, with no use of aggregate operations. Such views are called updatable views. This definition is oversimplified, but it captures the spirit of the restrictions. An update on such a restricted view can
always be implemented by updating the underlying base table in an unambiguous way. Consider the following view:

```
CREATE VIEW GoodStudents (sid, gpa)
AS SELECT S.sid, S.gpa
FROM Students S
WHERE S.gpa > 3.0
```

We can implement a command to modify the gpa of a GoodStudents row by modifying the corresponding row in Students. We can delete a GoodStudents row by deleting the corresponding row from Students. (In general, if the view did not include a key for the underlying table, several rows in the table could ‘correspond’ to a single row in the view. This would be the case, for example, if we used `S.sname` instead of `S.sid` in the definition of GoodStudents. A command that affects a row in the view would then affect all corresponding rows in the underlying table.)

We can insert a GoodStudents row by inserting a row into Students, using `null` values in columns of Students that do not appear in GoodStudents (e.g., `sname`, `login`). Note that primary key columns are not allowed to contain `null` values. Therefore, if we attempt to insert rows through a view that does not contain the primary key of the underlying table, the insertions will be rejected. For example, if GoodStudents contained `sname` but not `sid`, we could not insert rows into Students through insertions to GoodStudents.

An important observation is that an `INSERT` or `UPDATE` may change the underlying base table so that the resulting (i.e., inserted or modified) row is not in the view! For example, if we try to insert a row `(51234, 2.8)` into the view, this row can be (padded with `null` values in the other fields of Students and then) added to the underlying Students table, but it will not appear in the GoodStudents view because it does not satisfy the view condition `gpa > 3.0`. The SQL-92 default action is to allow this insertion, but we can disallow it by adding the clause `WITH CHECK OPTION` to the definition of the view.

We caution the reader that when a view is defined in terms of another view, the interaction between these view definitions with respect to updates and the `CHECK OPTION` clause can be complex; we will not go into the details.

**Need to Restrict View Updates**

While the SQL-92 rules on updatable views are more stringent than necessary, there are some fundamental problems with updates specified on views, and there is good reason to limit the class of views that can be updated. Consider the Students relation and a new relation called Clubs:
The Relational Model

Clubs\((\text{cname}: \text{string}, \text{jyear}: \text{date}, \text{mname}: \text{string})\)

A tuple in Clubs denotes that the student called \text{mname} has been a member of the club \text{cname} since the date \text{jyear}. Suppose that we are often interested in finding the names and logins of students with a \text{gpa} greater than 3 who belong to at least one club, along with the club name and the date they joined the club. We can define a view for this purpose:

\[
\text{CREATE VIEW ActiveStudents (name, login, club, since) AS }
\text{SELECT S.sname, S.login, C.cname, C.jyear }
\text{FROM Students S, Clubs C }
\text{WHERE S.sname = C.mname AND S.gpa > 3}
\]

Consider the instances of Students and Clubs shown in Figures 3.19 and 3.20. When evaluated using the instances \(C\) and \(S3\), \text{ActiveStudents} contains the rows shown in Figure 3.21.

\[
\begin{array}{|c|c|c|}
\hline
\text{name} & \text{login} & \text{club} \\
\hline
\text{Dave} & \text{dave@cs} & \text{Sailing} \\
\text{Smith} & \text{smith@ee} & \text{Hiking} \\
\text{Smith} & \text{smith@ee} & \text{Rowing} \\
\text{Smith} & \text{smith@math} & \text{Hiking} \\
\text{Smith} & \text{smith@math} & \text{Rowing} \\
\hline
\end{array}
\]

\text{Figure 3.21 Instance of ActiveStudents}

Now suppose that we want to delete the row \(\langle \text{Smith, smith@ee, Hiking, 1997} \rangle\) from \text{ActiveStudents}. How are we to do this? \text{ActiveStudents} rows are not stored explicitly but are computed as needed from the Students and Clubs tables using the view definition. So we must change either Students or Clubs (or both) in such a way that evaluating the

\text{We remark that Clubs has a poorly designed schema (chosen for the sake of our discussion of view updates), since it identifies students by name, which is not a candidate key for Students.}
view definition on the modified instance does not produce the row \((Smith, smith@ee, Hiking, 1997)\). This task can be accomplished in one of two ways: by either deleting the row \((53688, Smith, smith@ee, 18, 3.2)\) from Students or deleting the row \((Hiking, 1997, Smith)\) from Clubs. But neither solution is satisfactory. Removing the Students row has the effect of also deleting the row \((Smith, smith@ee, Rowing, 1998)\) from the view ActiveStudents. Removing the Clubs row has the effect of also deleting the row \((Smith, smith@math, Hiking, 1997)\) from the view ActiveStudents. Neither of these side effects is desirable. In fact, the only reasonable solution is to *disallow* such updates on views.

There are views involving more than one base table that can, in principle, be safely updated. The B-Students view that we introduced at the beginning of this section is an example of such a view. Consider the instance of B-Students shown in Figure 3.18 (with, of course, the corresponding instances of Students and Enrolled as in Figure 3.4). To insert a tuple, say \((Dave, 50000, Reggae203)\) B-Students, we can simply insert a tuple \((Reggae203, B, 50000)\) into Enrolled since there is already a tuple for sid 50000 in Students. To insert \((John, 55000, Reggae203)\), on the other hand, we have to insert \((Reggae203, B, 55000)\) into Enrolled and also insert \((55000, John, null, null, null)\) into Students. Observe how *null* values are used in fields of the inserted tuple whose value is not available. Fortunately, the view schema contains the primary key fields of both underlying base tables; otherwise, we would not be able to support insertions into this view. To delete a tuple from the view B-Students, we can simply delete the corresponding tuple from Enrolled.

Although this example illustrates that the SQL-92 rules on updatable views are unnecessarily restrictive, it also brings out the complexity of handling view updates in the general case. For practical reasons, the SQL-92 standard has chosen to allow only updates on a very restricted class of views.

### 3.7 DESTROYING/ALTERING TABLES AND VIEWS

If we decide that we no longer need a base table and want to destroy it (i.e., delete all the rows and remove the table definition information), we can use the **DROP TABLE** command. For example, **DROP TABLE Students** \texttt{RESTRICT} destroys the Students table unless some view or integrity constraint refers to Students; if so, the command fails. If the keyword \texttt{RESTRICT} is replaced by \texttt{CASCADE}, Students is dropped and any referencing views or integrity constraints are (recursively) dropped as well; one of these two keywords must always be specified. A view can be dropped using the **DROP VIEW** command, which is just like **DROP TABLE**.

**ALTER TABLE** modifies the structure of an existing table. To add a column called *maiden-name* to Students, for example, we would use the following command:
The definition of Students is modified to add this column, and all existing rows are padded with null values in this column. ALTER TABLE can also be used to delete columns and to add or drop integrity constraints on a table; we will not discuss these aspects of the command beyond remarking that dropping columns is treated very similarly to dropping tables or views.

3.8 POINTS TO REVIEW

- The main element of the relational model is a relation. A relation schema describes the structure of a relation by specifying the relation name and the names of each field. In addition, the relation schema includes domain constraints, which are type restrictions on the fields of the relation. The number of fields is called the degree of the relation. The relation instance is an actual table that contains a set of tuples that adhere to the relation schema. The number of tuples is called the cardinality of the relation. SQL-92 is a standard language for interacting with a DBMS. Its data definition language (DDL) enables the creation (CREATE TABLE) and modification (DELETE, UPDATE) of relations. (Section 3.1)

- Integrity constraints are conditions on a database schema that every legal database instance has to satisfy. Besides domain constraints, other important types of ICs are key constraints (a minimal set of fields that uniquely identify a tuple) and foreign key constraints (fields in one relation that refer to fields in another relation). SQL-92 supports the specification of the above kinds of ICs, as well as more general constraints called table constraints and assertions. (Section 3.2)

- ICs are enforced whenever a relation is modified and the specified ICs might conflict with the modification. For foreign key constraint violations, SQL-92 provides several alternatives to deal with the violation: NO ACTION, CASCADE, SET DEFAULT, and SET NULL. (Section 3.3)

- A relational database query is a question about the data. SQL supports a very expressive query language. (Section 3.4)

- There are standard translations of ER model constructs into SQL. Entity sets are mapped into relations. Relationship sets without constraints are also mapped into relations. When translating relationship sets with constraints, weak entity sets, class hierarchies, and aggregation, the mapping is more complicated. (Section 3.5)

- A view is a relation whose instance is not explicitly stored but is computed as needed. In addition to enabling logical data independence by defining the external schema through views, views play an important role in restricting access to data for

ALTER TABLE Students
ADD COLUMN maiden-name CHAR(10)
Chapter 3

security reasons. Since views might be defined through complex queries, handling updates specified on views is complicated, and SQL-92 has very stringent rules on when a view is updatable. (Section 3.6)

- SQL provides language constructs to modify the structure of tables (ALTER TABLE) and to destroy tables and views (DROP TABLE). (Section 3.7)

EXERCISES

Exercise 3.1 Define the following terms: relation schema, relational database schema, domain, relation instance, relation cardinality, and relation degree.

Exercise 3.2 How many distinct tuples are in a relation instance with cardinality 22?

Exercise 3.3 Does the relational model, as seen by an SQL query writer, provide physical and logical data independence? Explain.

Exercise 3.4 What is the difference between a candidate key and the primary key for a given relation? What is a superkey?

Exercise 3.5 Consider the instance of the Students relation shown in Figure 3.1.

1. Give an example of an attribute (or set of attributes) that you can deduce is not a candidate key, based on this instance being legal.

2. Is there any example of an attribute (or set of attributes) that you can deduce is a candidate key, based on this instance being legal?

Exercise 3.6 What is a foreign key constraint? Why are such constraints important? What is referential integrity?

Exercise 3.7 Consider the relations Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets in that were defined in Section 1.5.2.

1. List all the foreign key constraints among these relations.

2. Give an example of a (plausible) constraint involving one or more of these relations that is not a primary key or foreign key constraint.

Exercise 3.8 Answer each of the following questions briefly. The questions are based on the following relational schema:

\[
\begin{align*}
\text{Emp}(eid: \text{integer}, \ ename: \text{string}, \ age: \text{integer}, \ salary: \text{real}) \\
\text{Works}(eid: \text{integer}, \ did: \text{integer}, \ pct\_time: \text{integer}) \\
\text{Dept}(did: \text{integer}, \ dname: \text{string}, \ budget: \text{real}, \ managerid: \text{integer})
\end{align*}
\]

1. Give an example of a foreign key constraint that involves the Dept relation. What are the options for enforcing this constraint when a user attempts to delete a Dept tuple?
2. Write the SQL statements required to create the above relations, including appropriate versions of all primary and foreign key integrity constraints.

3. Define the Dept relation in SQL so that every department is guaranteed to have a manager.

4. Write an SQL statement to add ‘John Doe’ as an employee with $eid = 101$, $age = 32$ and $salary = 15,000$.

5. Write an SQL statement to give every employee a 10% raise.

6. Write an SQL statement to delete the ‘Toy’ department. Given the referential integrity constraints you chose for this schema, explain what happens when this statement is executed.

Exercise 3.9 Consider the SQL query whose answer is shown in Figure 3.6.

1. Modify this query so that only the $login$ column is included in the answer.

2. If the clause $WHERE \ S.gpa >= 2$ is added to the original query, what is the set of tuples in the answer?

Exercise 3.10 Explain why the addition of $NOT \ NULL$ constraints to the SQL definition of the Manages relation (in Section 3.5.3) would not enforce the constraint that each department must have a manager. What, if anything, is achieved by requiring that the $ssn$ field of Manages be non-null?

Exercise 3.11 Suppose that we have a ternary relationship $R$ between entity sets $A$, $B$, and $C$ such that $A$ has a key constraint and total participation and $B$ has a key constraint; these are the only constraints. $A$ has attributes $a_1$ and $a_2$, with $a_1$ being the key; $B$ and $C$ are similar. $R$ has no descriptive attributes. Write SQL statements that create tables corresponding to this information so as to capture as many of the constraints as possible. If you cannot capture some constraint, explain why.

Exercise 3.12 Consider the scenario from Exercise 2.2 where you designed an ER diagram for a university database. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Exercise 3.13 Consider the university database from Exercise 2.3 and the ER diagram that you designed. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Exercise 3.14 Consider the scenario from Exercise 2.4 where you designed an ER diagram for a company database. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Exercise 3.15 Consider the Notown database from Exercise 2.5. You have decided to recommend that Notown use a relational database system to store company data. Show the SQL statements for creating relations corresponding to the entity sets and relationship sets in your design. Identify any constraints in the ER diagram that you are unable to capture in the SQL statements and briefly explain why you could not express them.
Exercise 3.16 Translate your ER diagram from Exercise 2.6 into a relational schema, and show the SQL statements needed to create the relations, using only key and null constraints. If your translation cannot capture any constraints in the ER diagram, explain why.

In Exercise 2.6, you also modified the ER diagram to include the constraint that tests on a plane must be conducted by a technician who is an expert on that model. Can you modify the SQL statements defining the relations obtained by mapping the ER diagram to check this constraint?

Exercise 3.17 Consider the ER diagram that you designed for the Prescriptions-R-X chain of pharmacies in Exercise 2.7. Define relations corresponding to the entity sets and relationship sets in your design using SQL.

Exercise 3.18 Write SQL statements to create the corresponding relations to the ER diagram you designed for Exercise 2.8. If your translation cannot capture any constraints in the ER diagram, explain why.

PROJECT-BASED EXERCISES

Exercise 3.19 Create the relations Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets in Minibase.

Exercise 3.20 Insert the tuples shown in Figures 3.1 and 3.4 into the relations Students and Enrolled. Create reasonable instances of the other relations.

Exercise 3.21 What integrity constraints are enforced by Minibase?

Exercise 3.22 Run the SQL queries presented in this chapter.

BIBLIOGRAPHIC NOTES

The relational model was proposed in a seminal paper by Codd [156]. Childs [146] and Kuhns [392] foreshadowed some of these developments. Gallaire and Minker’s book [254] contains several papers on the use of logic in the context of relational databases. A system based on a variation of the relational model in which the entire database is regarded abstractly as a single relation, called the universal relation, is described in [655]. Extensions of the relational model to incorporate null values, which indicate an unknown or missing field value, are discussed by several authors; for example, [280, 335, 542, 662, 691]. Pioneering projects include System R [33, 129] at IBM San Jose Research Laboratory (now IBM Almaden Research Center), Ingres [628] at the University of California at Berkeley, PRTV [646] at the IBM UK Scientific Center in Peterlee, and QBE [702] at IBM T.J. Watson Research Center.

A rich theory underpins the field of relational databases. Texts devoted to theoretical aspects include those by Atzeni and DeAntonellis [38]; Maier [436]; and Abiteboul, Hull, and Vianu [3]. [355] is an excellent survey article.
Integrity constraints in relational databases have been discussed at length. [159] addresses semantic extensions to the relational model, but also discusses integrity, in particular referential integrity. [305] discusses semantic integrity constraints. [168] contains papers that address various aspects of integrity constraints, including in particular a detailed discussion of referential integrity. A vast literature deals with enforcing integrity constraints. [41] compares the cost of enforcing integrity constraints via compile-time, run-time, and post-execution checks. [124] presents an SQL-based language for specifying integrity constraints and identifies conditions under which integrity rules specified in this language can be violated. [624] discusses the technique of integrity constraint checking by query modification. [149] discusses real-time integrity constraints. Other papers on checking integrity constraints in databases include [69, 103, 117, 449]. [593] considers the approach of verifying the correctness of programs that access the database, instead of run-time checks. Note that this list of references is far from complete; in fact, it does not include any of the many papers on checking recursively specified integrity constraints. Some early papers in this widely studied area can be found in [254] and [253].

For references on SQL, see the bibliographic notes for Chapter 5. This book does not discuss specific products based on the relational model, but many fine books do discuss each of the major commercial systems; for example, Chamberlin’s book on DB2 [128], Date and McGoveran’s book on Sybase [172], and Koch and Loney’s book on Oracle [382].

Several papers consider the problem of translating updates specified on views into updates on the underlying table [49, 174, 360, 405, 683]. [250] is a good survey on this topic. See the bibliographic notes for Chapter 23 for references to work querying views and maintaining materialized views.

[642] discusses a design methodology based on developing an ER diagram and then translating to the relational model. Markowitz considers referential integrity in the context of ER to relational mapping and discusses the support provided in some commercial systems (as of that date) in [446, 447].
A memory is what is left when something happens and does not completely unhappen.

—Edward DeBono

This chapter initiates a study of the internals of an RDBMS. In terms of the DBMS architecture presented in Section 1.8, it covers the disk space manager, the buffer manager, and the layer that supports the abstraction of a file of records. Later chapters cover auxiliary structures to speed retrieval of desired subsets of the data, and the implementation of a relational query language.

Data in a DBMS is stored on storage devices such as disks and tapes; we concentrate on disks and cover tapes briefly. The disk space manager is responsible for keeping track of available disk space. The file manager, which provides the abstraction of a file of records to higher levels of DBMS code, issues requests to the disk space manager to obtain and relinquish space on disk. The file management layer requests and frees disk space in units of a page; the size of a page is a DBMS parameter, and typical values are 4 KB or 8 KB. The file management layer is responsible for keeping track of the pages in a file and for arranging records within pages.

When a record is needed for processing, it must be fetched from disk to main memory. The page on which the record resides is determined by the file manager. Sometimes, the file manager uses auxiliary data structures to quickly identify the page that contains a desired record. After identifying the required page, the file manager issues a request for the page to a layer of DBMS code called the buffer manager. The buffer manager fetches a requested page from disk into a region of main memory called the buffer pool and tells the file manager the location of the requested page.

We cover the above points in detail in this chapter. Section 7.1 introduces disks and tapes. Section 7.2 describes RAID disk systems. Section 7.3 discusses how a DBMS manages disk space, and Section 7.4 explains how a DBMS fetches data from disk into main memory. Section 7.5 discusses how a collection of pages is organized into a file and how auxiliary data structures can be built to speed up retrieval of records from a file. Section 7.6 covers different ways to arrange a collection of records on a page, and Section 7.7 covers alternative formats for storing individual records.
7.1 THE MEMORY HIERARCHY

Memory in a computer system is arranged in a hierarchy, as shown in Figure 7.1. At the top, we have primary storage, which consists of cache and main memory, and provides very fast access to data. Then comes secondary storage, which consists of slower devices such as magnetic disks. Tertiary storage is the slowest class of storage devices; for example, optical disks and tapes. Currently, the cost of a given amount of main memory is about 100 times the cost of the same amount of disk space, and tapes are even less expensive than disks. Slower storage devices such as tapes and disks play an important role in database systems because the amount of data is typically very large. Since buying enough main memory to store all data is prohibitively expensive, we must store data on tapes and disks and build database systems that can retrieve data from lower levels of the memory hierarchy into main memory as needed for processing.

There are reasons other than cost for storing data on secondary and tertiary storage. On systems with 32-bit addressing, only $2^{32}$ bytes can be directly referenced in main memory; the number of data objects may exceed this number! Further, data must be maintained across program executions. This requires storage devices that retain information when the computer is restarted (after a shutdown or a crash); we call such storage nonvolatile. Primary storage is usually volatile (although it is possible to make it nonvolatile by adding a battery backup feature), whereas secondary and tertiary storage is nonvolatile.

Tapes are relatively inexpensive and can store very large amounts of data. They are a good choice for archival storage, that is, when we need to maintain data for a long period but do not expect to access it very often. A Quantum DLT 4000 drive is a typical tape device; it stores 20 GB of data and can store about twice as much by compressing the data. It records data on 128 tape tracks, which can be thought of as a
linear sequence of adjacent bytes, and supports a sustained transfer rate of 1.5 MB/sec with uncompressed data (typically 3.0 MB/sec with compressed data). A single DLT 4000 tape drive can be used to access up to seven tapes in a stacked configuration, for a maximum compressed data capacity of about 280 GB.

The main drawback of tapes is that they are sequential access devices. We must essentially step through all the data in order and cannot directly access a given location on tape. For example, to access the last byte on a tape, we would have to wind through the entire tape first. This makes tapes unsuitable for storing operational data, or data that is frequently accessed. Tapes are mostly used to back up operational data periodically.

### 7.1.1 Magnetic Disks

Magnetic disks support direct access to a desired location and are widely used for database applications. A DBMS provides seamless access to data on disk; applications need not worry about whether data is in main memory or disk. To understand how disks work, consider Figure 7.2, which shows the structure of a disk in simplified form.

![Figure 7.2 Structure of a Disk](image)

Data is stored on disk in units called **disk blocks**. A disk block is a contiguous sequence of bytes and is the unit in which data is written to a disk and read from a disk. Blocks are arranged in concentric rings called **tracks**, on one or more **platters**. Tracks can be recorded on one or both surfaces of a platter; we refer to platters as
single-sided or double-sided accordingly. The set of all tracks with the same diameter is called a cylinder, because the space occupied by these tracks is shaped like a cylinder; a cylinder contains one track per platter surface. Each track is divided into arcs called sectors, whose size is a characteristic of the disk and cannot be changed. The size of a disk block can be set when the disk is initialized as a multiple of the sector size.

An array of disk heads, one per recorded surface, is moved as a unit; when one head is positioned over a block, the other heads are in identical positions with respect to their platters. To read or write a block, a disk head must be positioned on top of the block. As the size of a platter decreases, seek times also decrease since we have to move a disk head a smaller distance. Typical platter diameters are 3.5 inches and 5.25 inches.

Current systems typically allow at most one disk head to read or write at any one time. All the disk heads cannot read or write in parallel—this technique would increase data transfer rates by a factor equal to the number of disk heads, and considerably speed up sequential scans. The reason they cannot is that it is very difficult to ensure that all the heads are perfectly aligned on the corresponding tracks. Current approaches are both expensive and more prone to faults as compared to disks with a single active head. In practice very few commercial products support this capability, and then only in a limited way; for example, two disk heads may be able to operate in parallel.

A disk controller interfaces a disk drive to the computer. It implements commands to read or write a sector by moving the arm assembly and transferring data to and from the disk surfaces. A checksum is computed for when data is written to a sector and stored with the sector. The checksum is computed again when the data on the sector is read back. If the sector is corrupted or the read is faulty for some reason, it is very unlikely that the checksum computed when the sector is read matches the checksum computed when the sector was written. The controller computes checksums and if it detects an error, it tries to read the sector again. (Of course, it signals a failure if the sector is corrupted and read fails repeatedly.)

While direct access to any desired location in main memory takes approximately the same time, determining the time to access a location on disk is more complicated. The time to access a disk block has several components. Seek time is the time taken to move the disk heads to the track on which a desired block is located. Rotational delay is the waiting time for the desired block to rotate under the disk head; it is the time required for half a rotation on average and is usually less than seek time. Transfer time is the time to actually read or write the data in the block once the head is positioned, that is, the time for the disk to rotate over the block.
An example of a current disk: The IBM Deskstar 14GPX. The IBM Deskstar 14GPX is a 3.5 inch, 14.4 GB hard disk with an average seek time of 9.1 milliseconds (msec) and an average rotational delay of 4.17 msec. However, the time to seek from one track to the next is just 2.2 msec, the maximum seek time is 15.5 msec. The disk has five double-sided platters that spin at 7,200 rotations per minute. Each platter holds 3.35 GB of data, with a density of 2.6 gigabit per square inch. The data transfer rate is about 13 MB per second. To put these numbers in perspective, observe that a disk access takes about 10 msec, whereas accessing a main memory location typically takes less than 60 nanoseconds!

7.1.2 Performance Implications of Disk Structure

1. Data must be in memory for the DBMS to operate on it.

2. The unit for data transfer between disk and main memory is a block; if a single item on a block is needed, the entire block is transferred. Reading or writing a disk block is called an I/O (for input/output) operation.

3. The time to read or write a block varies, depending on the location of the data:
   \[ access\ time = seek\ time + rotational\ delay + transfer\ time \]

These observations imply that the time taken for database operations is affected significantly by how data is stored on disks. The time for moving blocks to or from disk usually dominates the time taken for database operations. To minimize this time, it is necessary to locate data records strategically on disk, because of the geometry and mechanics of disks. In essence, if two records are frequently used together, we should place them close together. The ‘closest’ that two records can be on a disk is to be on the same block. In decreasing order of closeness, they could be on the same track, the same cylinder, or an adjacent cylinder.

Two records on the same block are obviously as close together as possible, because they are read or written as part of the same block. As the platter spins, other blocks on the track being read or written rotate under the active head. In current disk designs, all the data on a track can be read or written in one revolution. After a track is read or written, another disk head becomes active, and another track in the same cylinder is read or written. This process continues until all tracks in the current cylinder are read or written, and then the arm assembly moves (in or out) to an adjacent cylinder. Thus, we have a natural notion of ‘closeness’ for blocks, which we can extend to a notion of \textit{next} and \textit{previous} blocks.

Exploiting this notion of \textit{next} by arranging records so that they are read or written sequentially is very important for reducing the time spent in disk I/Os. Sequential access minimizes seek time and rotational delay and is much faster than random access.
(This observation is reinforced and elaborated in Exercises 7.5 and 7.6, and the reader is urged to work through them.)

7.2 RAID

Disks are potential bottlenecks for system performance and storage system reliability. Even though disk performance has been improving continuously, microprocessor performance has advanced much more rapidly. The performance of microprocessors has improved at about 50 percent or more per year, but disk access times have improved at a rate of about 10 percent per year and disk transfer rates at a rate of about 20 percent per year. In addition, since disks contain mechanical elements, they have much higher failure rates than electronic parts of a computer system. If a disk fails, all the data stored on it is lost.

A disk array is an arrangement of several disks, organized so as to increase performance and improve reliability of the resulting storage system. Performance is increased through data striping. Data striping distributes data over several disks to give the impression of having a single large, very fast disk. Reliability is improved through redundancy. Instead of having a single copy of the data, redundant information is maintained. The redundant information is carefully organized so that in case of a disk failure, it can be used to reconstruct the contents of the failed disk. Disk arrays that implement a combination of data striping and redundancy are called redundant arrays of independent disks, or in short, RAID.\(^1\) Several RAID organizations, referred to as RAID levels, have been proposed. Each RAID level represents a different trade-off between reliability and performance.

In the remainder of this section, we will first discuss data striping and redundancy and then introduce the RAID levels that have become industry standards.

7.2.1 Data Striping

A disk array gives the user the abstraction of having a single, very large disk. If the user issues an I/O request, we first identify the set of physical disk blocks that store the data requested. These disk blocks may reside on a single disk in the array or may be distributed over several disks in the array. Then the set of blocks is retrieved from the disk(s) involved. Thus, how we distribute the data over the disks in the array influences how many disks are involved when an I/O request is processed.

\(^1\)Historically, the I in RAID stood for inexpensive, as a large number of small disks was much more economical than a single very large disk. Today, such very large disks are not even manufactured—a sign of the impact of RAID.
Redundancy schemes: Alternatives to the parity scheme include schemes based on Hamming codes and Reed-Solomon codes. In addition to recovery from single disk failures, Hamming codes can identify which disk has failed. Reed-Solomon codes can recover from up to two simultaneous disk failures. A detailed discussion of these schemes is beyond the scope of our discussion here; the bibliography provides pointers for the interested reader.

In data striping, the data is segmented into equal-size partitions that are distributed over multiple disks. The size of a partition is called the striping unit. The partitions are usually distributed using a round robin algorithm: If the disk array consists of $D$ disks, then partition $i$ is written onto disk $i \mod D$.

As an example, consider a striping unit of a bit. Since any $D$ successive data bits are spread over all $D$ data disks in the array, all I/O requests involve all disks in the array. Since the smallest unit of transfer from a disk is a block, each I/O request involves transfer of at least $D$ blocks. Since we can read the $D$ blocks from the $D$ disks in parallel, the transfer rate of each request is $D$ times the transfer rate of a single disk; each request uses the aggregated bandwidth of all disks in the array. But the disk access time of the array is basically the access time of a single disk since all disk heads have to move for all requests. Therefore, for a disk array with a striping unit of a single bit, the number of requests per time unit that the array can process and the average response time for each individual request are similar to that of a single disk.

As another example, consider a striping unit of a disk block. In this case, I/O requests of the size of a disk block are processed by one disk in the array. If there are many I/O requests of the size of a disk block and the requested blocks reside on different disks, we can process all requests in parallel and thus reduce the average response time of an I/O request. Since we distributed the striping partitions round-robin, large requests of the size of many contiguous blocks involve all disks. We can process the request by all disks in parallel and thus increase the transfer rate to the aggregated bandwidth of all $D$ disks.

7.2.2 Redundancy

While having more disks increases storage system performance, it also lowers overall storage system reliability. Assume that the mean-time-to-failure, or MTTF, of a single disk is 50,000 hours (about 5.7 years). Then, the MTTF of an array of 100 disks is only $50,000/100 = 500$ hours or about 21 days, assuming that failures occur independently and that the failure probability of a disk does not change over time. (Actually, disks have a higher failure probability early and late in their lifetimes. Early failures are often due to undetected manufacturing defects; late failures occur
since the disk wears out. Failures do not occur independently either: consider a fire in the building, an earthquake, or purchase of a set of disks that come from a ‘bad’ manufacturing batch.)

Reliability of a disk array can be increased by storing redundant information. If a disk failure occurs, the redundant information is used to reconstruct the data on the failed disk. Redundancy can immensely increase the MTTF of a disk array. When incorporating redundancy into a disk array design, we have to make two choices. First, we have to decide where to store the redundant information. We can either store the redundant information on a small number of check disks or we can distribute the redundant information uniformly over all disks.

The second choice we have to make is how to compute the redundant information. Most disk arrays store parity information: In the parity scheme, an extra check disk contains information that can be used to recover from failure of any one disk in the array. Assume that we have a disk array with \(D\) disks and consider the first bit on each data disk. Suppose that \(i\) of the \(D\) data bits are one. The first bit on the check disk is set to one if \(i\) is odd, otherwise it is set to zero. This bit on the check disk is called the parity of the data bits. The check disk contains parity information for each set of corresponding \(D\) data bits.

To recover the value of the first bit of a failed disk we first count the number of bits that are one on the \(D - 1\) nonfailed disks; let this number be \(j\). If \(j\) is odd and the parity bit is one, or if \(j\) is even and the parity bit is zero, then the value of the bit on the failed disk must have been zero. Otherwise, the value of the bit on the failed disk must have been one. Thus, with parity we can recover from failure of any one disk. Reconstruction of the lost information involves reading all data disks and the check disk.

For example, with an additional 10 disks with redundant information, the MTTF of our example storage system with 100 data disks can be increased to more than 250 years! What is more important, a large MTTF implies a small failure probability during the actual usage time of the storage system, which is usually much smaller than the reported lifetime or the MTTF. (Who actually uses 10-year-old disks?)

In a RAID system, the disk array is partitioned into reliability groups, where a reliability group consists of a set of data disks and a set of check disks. A common redundancy scheme (see box) is applied to each group. The number of check disks depends on the RAID level chosen. In the remainder of this section, we assume for ease of explanation that there is only one reliability group. The reader should keep in mind that actual RAID implementations consist of several reliability groups, and that the number of groups plays a role in the overall reliability of the resulting storage system.
7.2.3 Levels of Redundancy

Throughout the discussion of the different RAID levels, we consider sample data that would just fit on four disks. That is, without any RAID technology our storage system would consist of exactly four data disks. Depending on the RAID level chosen, the number of additional disks varies from zero to four.

**Level 0: Nonredundant**

A RAID Level 0 system uses data striping to increase the maximum bandwidth available. No redundant information is maintained. While being the solution with the lowest cost, reliability is a problem, since the MTTF decreases linearly with the number of disk drives in the array. RAID Level 0 has the best write performance of all RAID levels, because absence of redundant information implies that no redundant information needs to be updated! Interestingly, RAID Level 0 does not have the best read performance of all RAID levels, since systems with redundancy have a choice of scheduling disk accesses as explained in the next section.

In our example, the RAID Level 0 solution consists of only four data disks. Independent of the number of data disks, the effective space utilization for a RAID Level 0 system is always 100 percent.

**Level 1: Mirrored**

A RAID Level 1 system is the most expensive solution. Instead of having one copy of the data, two identical copies of the data on two different disks are maintained. This type of redundancy is often called **mirroring**. Every write of a disk block involves a write on both disks. These writes may not be performed simultaneously, since a global system failure (e.g., due to a power outage) could occur while writing the blocks and then leave both copies in an inconsistent state. Therefore, we always write a block on one disk first and then write the other copy on the mirror disk. Since two copies of each block exist on different disks, we can distribute reads between the two disks and allow **parallel reads** of different disk blocks that conceptually reside on the same disk. A read of a block can be scheduled to the disk that has the smaller expected access time. RAID Level 1 does not stripe the data over different disks, thus the transfer rate for a single request is comparable to the transfer rate of a single disk.

In our example, we need four data and four check disks with mirrored data for a RAID Level 1 implementation. The effective space utilization is 50 percent, independent of the number of data disks.
Level 0+1: Striping and Mirroring

RAID Level 0+1—sometimes also referred to as RAID level 10—combines striping and mirroring. Thus, as in RAID Level 1, read requests of the size of a disk block can be scheduled both to a disk or its mirror image. In addition, read requests of the size of several contiguous blocks benefit from the aggregated bandwidth of all disks. The cost for writes is analogous to RAID Level 1.

As in RAID Level 1, our example with four data disks requires four check disks and the effective space utilization is always 50 percent.

Level 2: Error-Correcting Codes

In RAID Level 2 the striping unit is a single bit. The redundancy scheme used is Hamming code. In our example with four data disks, only three check disks are needed. In general, the number of check disks grows logarithmically with the number of data disks.

Striping at the bit level has the implication that in a disk array with $D$ data disks, the smallest unit of transfer for a read is a set of $D$ blocks. Thus, Level 2 is good for workloads with many large requests since for each request the aggregated bandwidth of all data disks is used. But RAID Level 2 is bad for small requests of the size of an individual block for the same reason. (See the example in Section 7.2.1.) A write of a block involves reading $D$ blocks into main memory, modifying $D + C$ blocks and writing $D + C$ blocks to disk, where $C$ is the number of check disks. This sequence of steps is called a read-modify-write cycle.

For a RAID Level 2 implementation with four data disks, three check disks are needed. Thus, in our example the effective space utilization is about 57 percent. The effective space utilization increases with the number of data disks. For example, in a setup with 10 data disks, four check disks are needed and the effective space utilization is 71 percent. In a setup with 25 data disks, five check disks are required and the effective space utilization grows to 83 percent.

Level 3: Bit-Interleaved Parity

While the redundancy schema used in RAID Level 2 improves in terms of cost upon RAID Level 1, it keeps more redundant information than is necessary. Hamming code, as used in RAID Level 2, has the advantage of being able to identify which disk has failed. But disk controllers can easily detect which disk has failed. Thus, the check disks do not need to contain information to identify the failed disk. Information to recover the lost data is sufficient. Instead of using several disks to store Hamming code,
RAID Level 3 has a single check disk with parity information. Thus, the reliability overhead for RAID Level 3 is a single disk, the lowest overhead possible.

The performance characteristics of RAID Level 2 and RAID Level 3 are very similar. RAID Level 3 can also process only one I/O at a time, the minimum transfer unit is $D$ blocks, and a write requires a read-modify-write cycle.

**Level 4: Block-Interleaved Parity**

RAID Level 4 has a striping unit of a disk block, instead of a single bit as in RAID Level 3. Block-level striping has the advantage that read requests of the size of a disk block can be served entirely by the disk where the requested block resides. Large read requests of several disk blocks can still utilize the aggregated bandwidth of the $D$ disks.

The write of a single block still requires a read-modify-write cycle, but only one data disk and the check disk are involved. The parity on the check disk can be updated without reading all $D$ disk blocks, because the new parity can be obtained by noticing the differences between the old data block and the new data block and then applying the difference to the parity block on the check disk:

$$\text{NewParity} = (\text{OldData XOR NewData}) \text{ XOR OldParity}$$

The read-modify-write cycle involves reading of the old data block and the old parity block, modifying the two blocks, and writing them back to disk, resulting in four disk accesses per write. Since the check disk is involved in each write, it can easily become the bottleneck.

RAID Level 3 and 4 configurations with four data disks require just a single check disk. Thus, in our example, the effective space utilization is 80 percent. The effective space utilization increases with the number of data disks, since always only one check disk is necessary.

**Level 5: Block-Interleaved Distributed Parity**

RAID Level 5 improves upon Level 4 by distributing the parity blocks uniformly over all disks, instead of storing them on a single check disk. This distribution has two advantages. First, several write requests can potentially be processed in parallel, since the bottleneck of a unique check disk has been eliminated. Second, read requests have a higher level of parallelism. Since the data is distributed over all disks, read requests involve all disks, whereas in systems with a dedicated check disk the check disk never participates in reads.
A RAID Level 5 system has the best performance of all RAID levels with redundancy for small and large read and large write requests. Small writes still require a read-modify-write cycle and are thus less efficient than in RAID Level 1.

In our example, the corresponding RAID Level 5 system has 5 disks overall and thus the effective space utilization is the same as in RAID levels 3 and 4.

**Level 6: P+Q Redundancy**

The motivation for RAID Level 6 is the observation that recovery from failure of a single disk is not sufficient in very large disk arrays. First, in large disk arrays, a second disk might fail before replacement of an already failed disk could take place. In addition, the probability of a disk failure during recovery of a failed disk is not negligible.

A RAID Level 6 system uses Reed-Solomon codes to be able to recover from up to two simultaneous disk failures. RAID Level 6 requires (conceptually) two check disks, but it also uniformly distributes redundant information at the block level as in RAID Level 5. Thus, the performance characteristics for small and large read requests and for large write requests are analogous to RAID Level 5. For small writes, the read-modify-write procedure involves six instead of four disks as compared to RAID Level 5, since two blocks with redundant information need to be updated.

For a RAID Level 6 system with storage capacity equal to four data disks, six disks are required. Thus, in our example, the effective space utilization is 66 percent.

### 7.2.4 Choice of RAID Levels

If data loss is not an issue, RAID Level 0 improves overall system performance at the lowest cost. RAID Level 0+1 is superior to RAID Level 1. The main application areas for RAID Level 0+1 systems are small storage subsystems where the cost of mirroring is moderate. Sometimes RAID Level 0+1 is used for applications that have a high percentage of writes in their workload, since RAID Level 0+1 provides the best write performance. RAID levels 2 and 4 are always inferior to RAID levels 3 and 5, respectively. RAID Level 3 is appropriate for workloads consisting mainly of large transfer requests of several contiguous blocks. The performance of a RAID Level 3 system is bad for workloads with many small requests of a single disk block. RAID Level 5 is a good general-purpose solution. It provides high performance for large requests as well as for small requests. RAID Level 6 is appropriate if a higher level of reliability is required.
7.3 DISK SPACE MANAGEMENT

The lowest level of software in the DBMS architecture discussed in Section 1.8, called the disk space manager, manages space on disk. Abstractly, the disk space manager supports the concept of a page as a unit of data, and provides commands to allocate or deallocate a page and read or write a page. The size of a page is chosen to be the size of a disk block and pages are stored as disk blocks so that reading or writing a page can be done in one disk I/O.

It is often useful to allocate a sequence of pages as a contiguous sequence of blocks to hold data that is frequently accessed in sequential order. This capability is essential for exploiting the advantages of sequentially accessing disk blocks, which we discussed earlier in this chapter. Such a capability, if desired, must be provided by the disk space manager to higher-level layers of the DBMS.

Thus, the disk space manager hides details of the underlying hardware (and possibly the operating system) and allows higher levels of the software to think of the data as a collection of pages.

7.3.1 Keeping Track of Free Blocks

A database grows and shrinks as records are inserted and deleted over time. The disk space manager keeps track of which disk blocks are in use, in addition to keeping track of which pages are on which disk blocks. Although it is likely that blocks are initially allocated sequentially on disk, subsequent allocations and deallocations could in general create ‘holes.’

One way to keep track of block usage is to maintain a list of free blocks. As blocks are deallocated (by the higher-level software that requests and uses these blocks), we can add them to the free list for future use. A pointer to the first block on the free block list is stored in a known location on disk.

A second way is to maintain a bitmap with one bit for each disk block, which indicates whether a block is in use or not. A bitmap also allows very fast identification and allocation of contiguous areas on disk. This is difficult to accomplish with a linked list approach.

7.3.2 Using OS File Systems to Manage Disk Space

Operating systems also manage space on disk. Typically, an operating system supports the abstraction of a file as a sequence of bytes. The OS manages space on the disk and translates requests such as “Read byte i of file f” into corresponding low-level
Many database systems do not rely on the OS file system and instead do their own disk management, either from scratch or by extending OS facilities. The reasons are practical as well as technical. One practical reason is that a DBMS vendor who wishes to support several OS platforms cannot assume features specific to any OS, for portability, and would therefore try to make the DBMS code as self-contained as possible. A technical reason is that on a 32-bit system, the largest file size is 4 GB, whereas a DBMS may want to access a single file larger than that. A related problem is that typical OS files cannot span disk devices, which is often desirable or even necessary in a DBMS. Additional technical reasons why a DBMS does not rely on the OS file system are outlined in Section 7.4.2.

7.4 BUFFER MANAGER

To understand the role of the buffer manager, consider a simple example. Suppose that the database contains 1,000,000 pages, but only 1,000 pages of main memory are available for holding data. Consider a query that requires a scan of the entire file. Because all the data cannot be brought into main memory at one time, the DBMS must bring pages into main memory as they are needed and, in the process, decide what existing page in main memory to replace to make space for the new page. The policy used to decide which page to replace is called the replacement policy.

In terms of the DBMS architecture presented in Section 1.8, the buffer manager is the software layer that is responsible for bringing pages from disk to main memory as needed. The buffer manager manages the available main memory by partitioning it into a collection of pages, which we collectively refer to as the buffer pool. The main memory pages in the buffer pool are called frames; it is convenient to think of them as slots that can hold a page (that usually resides on disk or other secondary storage media).

Higher levels of the DBMS code can be written without worrying about whether data pages are in memory or not; they ask the buffer manager for the page, and it is brought into a frame in the buffer pool if it is not already there. Of course, the higher-level code that requests a page must also release the page when it is no longer needed, by informing the buffer manager, so that the frame containing the page can be reused. The higher-level code must also inform the buffer manager if it modifies the requested page; the buffer manager then makes sure that the change is propagated to the copy of the page on disk. Buffer management is illustrated in Figure 7.3.
In addition to the buffer pool itself, the buffer manager maintains some bookkeeping information, and two variables for each frame in the pool: pin\_count and dirty. The number of times that the page currently in a given frame has been requested but not released—the number of current users of the page—is recorded in the pin\_count variable for that frame. The boolean variable dirty indicates whether the page has been modified since it was brought into the buffer pool from disk.

Initially, the pin\_count for every frame is set to 0, and the dirty bits are turned off. When a page is requested the buffer manager does the following:

1. Checks the buffer pool to see if some frame contains the requested page, and if so increments the pin\_count of that frame. If the page is not in the pool, the buffer manager brings it in as follows:

   (a) Chooses a frame for replacement, using the replacement policy, and increments its pin\_count.

   (b) If the dirty bit for the replacement frame is on, writes the page it contains to disk (that is, the disk copy of the page is overwritten with the contents of the frame).

   (c) Reads the requested page into the replacement frame.

2. Returns the (main memory) address of the frame containing the requested page to the requestor.
Incrementing \textit{pin\_count} is often called \textbf{pinning} the requested page in its frame. When the code that calls the buffer manager and requests the page subsequently calls the buffer manager and releases the page, the \textit{pin\_count} of the frame containing the requested page is decremented. This is called \textbf{unpinning} the page. If the requestor has modified the page, it also informs the buffer manager of this at the time that it unpins the page, and the \textit{dirty} bit for the frame is set. The buffer manager will not read another page into a frame until its \textit{pin\_count} becomes 0, that is, until all requestors of the page have unpinned it.

If a requested page is not in the buffer pool, and if a free frame is not available in the buffer pool, a frame with \textit{pin\_count} 0 is chosen for replacement. If there are many such frames, a frame is chosen according to the buffer manager’s replacement policy. We discuss various replacement policies in Section 7.4.1.

When a page is eventually chosen for replacement, if the \textit{dirty} bit is not set, it means that the page has not been modified since being brought into main memory. Thus, there is no need to write the page back to disk; the copy on disk is identical to the copy in the frame, and the frame can simply be overwritten by the newly requested page. Otherwise, the modifications to the page must be propagated to the copy on disk. (The crash recovery protocol may impose further restrictions, as we saw in Section 1.7. For example, in the Write-Ahead Log (WAL) protocol, special log records are used to describe the changes made to a page. The log records pertaining to the page that is to be replaced may well be in the buffer; if so, the protocol requires that they be written to disk \textit{before} the page is written to disk.)

If there is no page in the buffer pool with \textit{pin\_count} 0 and a page that is not in the pool is requested, the buffer manager must wait until some page is released before responding to the page request. In practice, the transaction requesting the page may simply be aborted in this situation! So pages should be released—by the code that calls the buffer manager to request the page—as soon as possible.

A good question to ask at this point is “What if a page is requested by several different transactions?” That is, what if the page is requested by programs executing independently on behalf of different users? There is the potential for such programs to make conflicting changes to the page. The locking protocol (enforced by higher-level DBMS code, in particular the transaction manager) ensures that each transaction obtains a shared or exclusive lock before requesting a page to read or modify. Two different transactions cannot hold an exclusive lock on the same page at the same time; this is how conflicting changes are prevented. The buffer manager simply assumes that the appropriate lock has been obtained before a page is requested.
7.4.1 Buffer Replacement Policies

The policy that is used to choose an unpinned page for replacement can affect the time taken for database operations considerably. Many alternative policies exist, and each is suitable in different situations.

The best-known replacement policy is least recently used (LRU). This can be implemented in the buffer manager using a queue of pointers to frames with pin_count 0. A frame is added to the end of the queue when it becomes a candidate for replacement (that is, when the pin_count goes to 0). The page chosen for replacement is the one in the frame at the head of the queue.

A variant of LRU, called clock replacement, has similar behavior but less overhead. The idea is to choose a page for replacement using a current variable that takes on values 1 through N, where N is the number of buffer frames, in circular order. We can think of the frames being arranged in a circle, like a clock’s face, and current as a clock hand moving across the face. In order to approximate LRU behavior, each frame also has an associated referenced bit, which is turned on when the page pin_count goes to 0.

The current frame is considered for replacement. If the frame is not chosen for replacement, current is incremented and the next frame is considered; this process is repeated until some frame is chosen. If the current frame has pin_count greater than 0, then it is not a candidate for replacement and current is incremented. If the current frame has the referenced bit turned on, the clock algorithm turns the referenced bit off and increments current—this way, a recently referenced page is less likely to be replaced. If the current frame has pin_count 0 and its referenced bit is off, then the page in it is chosen for replacement. If all frames are pinned in some sweep of the clock hand (that is, the value of current is incremented until it repeats), this means that no page in the buffer pool is a replacement candidate.

The LRU and clock policies are not always the best replacement strategies for a database system, particularly if many user requests require sequential scans of the data. Consider the following illustrative situation. Suppose the buffer pool has 10 frames, and the file to be scanned has 10 or fewer pages. Assuming, for simplicity, that there are no competing requests for pages, only the first scan of the file does any I/O. Page requests in subsequent scans will always find the desired page in the buffer pool. On the other hand, suppose that the file to be scanned has 11 pages (which is one more than the number of available pages in the buffer pool). Using LRU, every scan of the file will result in reading every page of the file! In this situation, called sequential flooding, LRU is the worst possible replacement strategy.
Buffer management in practice: IBM DB2 and Sybase ASE allow buffers to be partitioned into named pools. Each database, table, or index can be bound to one of these pools. Each pool can be configured to use either LRU or clock replacement in ASE; DB2 uses a variant of clock replacement, with the initial clock value based on the nature of the page (e.g., index nonleaves get a higher starting clock value, which delays their replacement). Interestingly, a buffer pool client in DB2 can explicitly indicate that it *hates* a page, making the page the next choice for replacement. As a special case, DB2 applies MRU for the pages fetched in some utility operations (e.g., RUNSTATS), and DB2 V6 also supports FIFO. Informix and Oracle 7 both maintain a single global buffer pool using LRU; Microsoft SQL Server has a single pool using clock replacement. In Oracle 8, tables can be bound to one of two pools; one has high priority, and the system attempts to keep pages in this pool in memory.

Beyond setting a maximum number of pins for a given transaction, there are typically no features for controlling buffer pool usage on a per-transaction basis. Microsoft SQL Server, however, supports a reservation of buffer pages by queries that require large amounts of memory (e.g., queries involving sorting or hashing).

Other replacement policies include first in first out (FIFO) and most recently used (MRU), which also entail overhead similar to LRU, and random, among others. The details of these policies should be evident from their names and the preceding discussion of LRU and clock.

7.4.2 Buffer Management in DBMS versus OS

Obvious similarities exist between virtual memory in operating systems and buffer management in database management systems. In both cases the goal is to provide access to more data than will fit in main memory, and the basic idea is to bring in pages from disk to main memory as needed, replacing pages that are no longer needed in main memory. Why can’t we build a DBMS using the virtual memory capability of an OS? A DBMS can often predict the order in which pages will be accessed, or page reference patterns, much more accurately than is typical in an OS environment, and it is desirable to utilize this property. Further, a DBMS needs more control over when a page is written to disk than an OS typically provides.

A DBMS can often predict reference patterns because most page references are generated by higher-level operations (such as sequential scans or particular implementations of various relational algebra operators) with a known pattern of page accesses. This ability to predict reference patterns allows for a better choice of pages to replace and makes the idea of specialized buffer replacement policies more attractive in the DBMS environment.
**Prefetching:** In IBM DB2, both sequential and list prefetch (prefetching a list of pages) are supported. In general, the prefetch size is 32 4KB pages, but this can be set by the user. For some sequential type database utilities (e.g., COPY, RUNSTATS), DB2 will prefetch up to 64 4KB pages. For a smaller buffer pool (i.e., less than 1000 buffers), the prefetch quantity is adjusted downward to 16 or 8 pages. Prefetch size can be configured by the user; for certain environments, it may be best to prefetch 1000 pages at a time! Sybase ASE supports asynchronous prefetching of up to 256 pages, and uses this capability to reduce latency during indexed access to a table in a range scan. Oracle 8 uses prefetching for sequential scan, retrieving large objects, and for certain index scans. Microsoft SQL Server supports prefetching for sequential scan and for scans along the leaf level of a B+ tree index and the prefetch size can be adjusted as a scan progresses. SQL Server also uses asynchronous prefetching extensively. Informix supports prefetching with a user-defined prefetch size.

Even more important, being able to predict reference patterns enables the use of a simple and very effective strategy called **prefetching of pages**. The buffer manager can anticipate the next several page requests and fetch the corresponding pages into memory before the pages are requested. This strategy has two benefits. First, the pages are available in the buffer pool when they are requested. Second, reading in a contiguous block of pages is much faster than reading the same pages at different times in response to distinct requests. (Review the discussion of disk geometry to appreciate why this is so.) If the pages to be prefetched are not contiguous, recognizing that several pages need to be fetched can nonetheless lead to faster I/O because an order of retrieval can be chosen for these pages that minimizes seek times and rotational delays.

Incidentally, note that the I/O can typically be done concurrently with CPU computation. Once the prefetch request is issued to the disk, the disk is responsible for reading the requested pages into memory pages and the CPU can continue to do other work.

A DBMS also requires the ability to explicitly **force** a page to disk, that is, to ensure that the copy of the page on disk is updated with the copy in memory. As a related point, a DBMS must be able to ensure that certain pages in the buffer pool are written to disk before certain other pages are written, in order to implement the WAL protocol for crash recovery, as we saw in Section 1.7. Virtual memory implementations in operating systems cannot be relied upon to provide such control over when pages are written to disk; the OS command to write a page to disk may be implemented by essentially recording the write request, and deferring the actual modification of the disk copy. If the system crashes in the interim, the effects can be catastrophic for a DBMS. (Crash recovery is discussed further in Chapter 20.)
7.5 FILES AND INDEXES

We now turn our attention from the way pages are stored on disk and brought into main memory to the way pages are used to store records and organized into logical collections or files. Higher levels of the DBMS code treat a page as effectively being a collection of records, ignoring the representation and storage details. In fact, the concept of a collection of records is not limited to the contents of a single page; a file of records is a collection of records that may reside on several pages. In this section, we consider how a collection of pages can be organized as a file. We discuss how the space on a page can be organized to store a collection of records in Sections 7.6 and 7.7.

Each record has a unique identifier called a record id, or rid for short. As we will see in Section 7.6, we can identify the page containing a record by using the record’s rid. The basic file structure that we consider, called a heap file, stores records in random order and supports retrieval of all records or retrieval of a particular record specified by its rid. Sometimes we want to retrieve records by specifying some condition on the fields of desired records, for example, “Find all employee records with age 35.” To speed up such selections, we can build auxiliary data structures that allow us to quickly find the rids of employee records that satisfy the given selection condition. Such an auxiliary structure is called an index; we introduce indexes in Section 7.5.2.

7.5.1 Heap Files

The simplest file structure is an unordered file or heap file. The data in the pages of a heap file is not ordered in any way, and the only guarantee is that one can retrieve all records in the file by repeated requests for the next record. Every record in the file has a unique rid, and every page in a file is of the same size.

Supported operations on a heap file include create and destroy files, insert a record, delete a record with a given rid, get a record with a given rid, and scan all records in the file. To get or delete a record with a given rid, note that we must be able to find the id of the page containing the record, given the id of the record.

We must keep track of the pages in each heap file in order to support scans, and we must keep track of pages that contain free space in order to implement insertion efficiently. We discuss two alternative ways to maintain this information. In each of these alternatives, pages must hold two pointers (which are page ids) for file-level bookkeeping in addition to the data.
Linked List of Pages

One possibility is to maintain a heap file as a doubly linked list of pages. The DBMS can remember where the first page is located by maintaining a table containing pairs of \(\langle\text{heap\_file\_name}, \text{page\_1\_addr}\rangle\) in a known location on disk. We call the first page of the file the header page.

An important task is to maintain information about empty slots created by deleting a record from the heap file. This task has two distinct parts: how to keep track of free space within a page and how to keep track of pages that have some free space. We consider the first part in Section 7.6. The second part can be addressed by maintaining a doubly linked list of pages with free space and a doubly linked list of full pages; together, these lists contain all pages in the heap file. This organization is illustrated in Figure 7.4; note that each pointer is really a page id.

If a new page is required, it is obtained by making a request to the disk space manager and then added to the list of pages in the file (probably as a page with free space, because it is unlikely that the new record will take up all the space on the page). If a page is to be deleted from the heap file, it is removed from the list and the disk space manager is told to deallocate it. (Note that the scheme can easily be generalized to allocate or deallocate a sequence of several pages and maintain a doubly linked list of these page sequences.)

One disadvantage of this scheme is that virtually all pages in a file will be on the free list if records are of variable length, because it is likely that every page has at least a few free bytes. To insert a typical record, we must retrieve and examine several pages on the free list before we find one with enough free space. The directory-based heap file organization that we discuss next addresses this problem.
Directory of Pages

An alternative to a linked list of pages is to maintain a directory of pages. The DBMS must remember where the first directory page of each heap file is located. The directory is itself a collection of pages and is shown as a linked list in Figure 7.5. (Other organizations are possible for the directory itself, of course.)

![Figure 7.5 Heap File Organization with a Directory](image)

Each directory entry identifies a page (or a sequence of pages) in the heap file. As the heap file grows or shrinks, the number of entries in the directory—and possibly the number of pages in the directory itself—grows or shrinks correspondingly. Note that since each directory entry is quite small in comparison to a typical page, the size of the directory is likely to be very small in comparison to the size of the heap file.

Free space can be managed by maintaining a bit per entry, indicating whether the corresponding page has any free space, or a count per entry, indicating the amount of free space on the page. If the file contains variable-length records, we can examine the free space count for an entry to determine if the record will fit on the page pointed to by the entry. Since several entries fit on a directory page, we can efficiently search for a data page with enough space to hold a record that is to be inserted.

7.5.2 Introduction to Indexes

Sometimes we want to find all records that have a given value in a particular field. If we can find the rids of all such records, we can locate the page containing each record from the record’s rid; however, the heap file organization does not help us to find the
rids of such records. An index is an auxiliary data structure that is intended to help us find rids of records that meet a selection condition.

Consider how you locate a desired book in a library. You can search a collection of index cards, sorted on author name or book title, to find the call number for the book. Because books are stored according to call numbers, the call number enables you to walk to the shelf that contains the book you need. Observe that an index on author name cannot be used to locate a book by title, and vice versa; each index speeds up certain kinds of searches, but not all. This is illustrated in Figure 7.6.

The same ideas apply when we want to support efficient retrieval of a desired subset of the data in a file. From an implementation standpoint, an index is just another kind of file, containing records that direct traffic on requests for data records. Every index has an associated search key, which is a collection of one or more fields of the file of records for which we are building the index; any subset of the fields can be a search key. We sometimes refer to the file of records as the indexed file.

An index is designed to speed up equality or range selections on the search key. For example, if we wanted to build an index to improve the efficiency of queries about employees of a given age, we could build an index on the age attribute of the employee dataset. The records stored in an index file, which we refer to as entries to avoid confusion with data records, allow us to find data records with a given search key value. In our example the index might contain \( \langle \text{age}, \text{rid} \rangle \) pairs, where rid identifies a data record.

The pages in the index file are organized in some way that allows us to quickly locate those entries in the index that have a given search key value. For example, we have to find entries with \( \text{age} \geq 30 \) (and then follow the rids in the retrieved entries) in order to find employee records for employees who are older than 30. Organization techniques, or data structures, for index files are called access methods, and several are known,
Rids in commercial systems: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all implement record ids as a page id and slot number. Sybase ASE uses the following page organization, which is typical: Pages contain a header followed by the rows and a slot array. The header contains the page identity, its allocation state, page free space state, and a timestamp. The slot array is simply a mapping of slot number to page offset.

Oracle 8 and SQL Server use logical record ids rather than page id and slot number in one special case: If a table has a clustered index, then records in the table are identified using the key value for the clustered index. This has the advantage that secondary indexes don’t have to be reorganized if records are moved across pages.

including B+ trees (Chapter 9) and hash-based structures (Chapter 10). B+ tree index files and hash-based index files are built using the page allocation and manipulation facilities provided by the disk space manager, just like heap files.

7.6 PAGE FORMATS *

The page abstraction is appropriate when dealing with I/O issues, but higher levels of the DBMS see data as a collection of records. In this section, we consider how a collection of records can be arranged on a page. We can think of a page as a collection of slots, each of which contains a record. A record is identified by using the pair \( \langle \text{page id}, \text{slot number} \rangle \); this is the record id (rid). (We remark that an alternative way to identify records is to assign each record a unique integer as its rid and to maintain a table that lists the page and slot of the corresponding record for each rid. Due to the overhead of maintaining this table, the approach of using \( \langle \text{page id}, \text{slot number} \rangle \) as an rid is more common.)

We now consider some alternative approaches to managing slots on a page. The main considerations are how these approaches support operations such as searching, inserting, or deleting records on a page.

7.6.1 Fixed-Length Records

If all records on the page are guaranteed to be of the same length, record slots are uniform and can be arranged consecutively within a page. At any instant, some slots are occupied by records, and others are unoccupied. When a record is inserted into the page, we must locate an empty slot and place the record there. The main issues are how we keep track of empty slots and how we locate all records on a page. The alternatives hinge on how we handle the deletion of a record.
The first alternative is to store records in the first $N$ slots (where $N$ is the number of records on the page); whenever a record is deleted, we move the last record on the page into the vacated slot. This format allows us to locate the $i$th record on a page by a simple offset calculation, and all empty slots appear together at the end of the page. However, this approach does not work if there are external references to the record that is moved (because the rid contains the slot number, which is now changed).

The second alternative is to handle deletions by using an array of bits, one per slot, to keep track of free slot information. Locating records on the page requires scanning the bit array to find slots whose bit is on; when a record is deleted, its bit is turned off. The two alternatives for storing fixed-length records are illustrated in Figure 7.7. Note that in addition to the information about records on the page, a page usually contains additional file-level information (e.g., the id of the next page in the file). The figure does not show this additional information.

![Figure 7.7](image)

The slotted page organization described for variable-length records in Section 7.6.2 can also be used for fixed-length records. It becomes attractive if we need to move records around on a page for reasons other than keeping track of space freed by deletions. A typical example is that we want to keep the records on a page sorted (according to the value in some field).

### 7.6.2 Variable-Length Records

If records are of variable length, then we cannot divide the page into a fixed collection of slots. The problem is that when a new record is to be inserted, we have to find an empty slot of just the right length—if we use a slot that is too big, we waste space, and obviously we cannot use a slot that is smaller than the record length. Therefore, when a record is inserted, we must allocate just the right amount of space for it, and when a record is deleted, we must move records to fill the hole created by the deletion,
in order to ensure that all the free space on the page is contiguous. Thus, the ability to move records on a page becomes very important.

The most flexible organization for variable-length records is to maintain a directory of slots for each page, with a \( (\text{record offset}, \text{record length}) \) pair per slot. The first component (\text{record offset}) is a 'pointer' to the record, as shown in Figure 7.8; it is the offset in bytes from the start of the data area on the page to the start of the record. Deletion is readily accomplished by setting the record offset to -1. Records can be moved around on the page because the rid, which is the page number and slot number (that is, position in the directory), does not change when the record is moved; only the record offset stored in the slot changes.

![Figure 7.8 Page Organization for Variable-Length Records](image)

The space available for new records must be managed carefully because the page is not preformatted into slots. One way to manage free space is to maintain a pointer (that is, offset from the start of the data area on the page) that indicates the start of the free space area. When a new record is too large to fit into the remaining free space, we have to move records on the page to reclaim the space freed by records that have been deleted earlier. The idea is to ensure that after reorganization, all records appear contiguously, followed by the available free space.

A subtle point to be noted is that the slot for a deleted record cannot always be removed from the slot directory, because slot numbers are used to identify records—by deleting a slot, we change (decrement) the slot number of subsequent slots in the slot directory, and thereby change the rid of records pointed to by subsequent slots. The
only way to remove slots from the slot directory is to remove the last slot if the record that it points to is deleted. However, when a record is inserted, the slot directory should be scanned for an element that currently does not point to any record, and this slot should be used for the new record. A new slot is added to the slot directory only if all existing slots point to records. If inserts are much more common than deletes (as is typically the case), the number of entries in the slot directory is likely to be very close to the actual number of records on the page.

This organization is also useful for fixed-length records if we need to move them around frequently; for example, when we want to maintain them in some sorted order. Indeed, when all records are the same length, instead of storing this common length information in the slot for each record, we can store it once in the system catalog.

In some special situations (e.g., the internal pages of a B+ tree, which we discuss in Chapter 9), we may not care about changing the rid of a record. In this case the slot directory can be compacted after every record deletion; this strategy guarantees that the number of entries in the slot directory is the same as the number of records on the page. If we do not care about modifying rids, we can also sort records on a page in an efficient manner by simply moving slot entries rather than actual records, which are likely to be much larger than slot entries.

A simple variation on the slotted organization is to maintain only record offsets in the slots. For variable-length records, the length is then stored with the record (say, in the first bytes). This variation makes the slot directory structure for pages with fixed-length records be the same as for pages with variable-length records.

### 7.7 RECORD FORMATS *

In this section we discuss how to organize fields within a record. While choosing a way to organize the fields of a record, we must take into account whether the fields of the record are of fixed or variable length and consider the cost of various operations on the record, including retrieval and modification of fields.

Before discussing record formats, we note that in addition to storing individual records, information that is common to all records of a given record type (such as the number of fields and field types) is stored in the system catalog, which can be thought of as a description of the contents of a database, maintained by the DBMS (Section 13.2). This avoids repeated storage of the same information with each record of a given type.
Record formats in commercial systems: In IBM DB2, fixed length fields are at fixed offsets from the beginning of the record. Variable length fields have offset and length in the fixed offset part of the record, and the fields themselves follow the fixed length part of the record. Informix, Microsoft SQL Server, and Sybase ASE use the same organization with minor variations. In Oracle 8, records are structured as if all fields are potentially variable length; a record is a sequence of length–data pairs, with a special length value used to denote a null value.

### 7.7.1 Fixed-Length Records

In a fixed-length record, each field has a fixed length (that is, the value in this field is of the same length in all records), and the number of fields is also fixed. The fields of such a record can be stored consecutively, and, given the address of the record, the address of a particular field can be calculated using information about the lengths of preceding fields, which is available in the system catalog. This record organization is illustrated in Figure 7.9.

![Figure 7.9 Organization of Records with Fixed-Length Fields](image)

### 7.7.2 Variable-Length Records

In the relational model, every record in a relation contains the same number of fields. If the number of fields is fixed, a record is of variable length only because some of its fields are of variable length.

One possible organization is to store fields consecutively, separated by delimiters (which are special characters that do not appear in the data itself). This organization requires a scan of the record in order to locate a desired field.

An alternative is to reserve some space at the beginning of a record for use as an array of integer offsets—the i-th integer in this array is the starting address of the i-th field value relative to the start of the record. Note that we also store an offset to the end of the record; this offset is needed to recognize where the last field ends. Both alternatives are illustrated in Figure 7.10.
Storing Data: Disks and Files

The second approach is typically superior. For the overhead of the offset array, we get direct access to any field. We also get a clean way to deal with null values. A null value is a special value used to denote that the value for a field is unavailable or inapplicable. If a field contains a null value, the pointer to the end of the field is set to be the same as the pointer to the beginning of the field. That is, no space is used for representing the null value, and a comparison of the pointers to the beginning and the end of the field is used to determine that the value in the field is null.

Variable-length record formats can obviously be used to store fixed-length records as well; sometimes, the extra overhead is justified by the added flexibility, because issues such as supporting null values and adding fields to a record type arise with fixed-length records as well.

Having variable-length fields in a record can raise some subtle issues, especially when a record is modified.

- Modifying a field may cause it to grow, which requires us to shift all subsequent fields to make space for the modification in all three record formats presented above.

- A record that is modified may no longer fit into the space remaining on its page. If so, it may have to be moved to another page. If rids, which are used to ‘point’ to a record, include the page number (see Section 7.6), moving a record to another page causes a problem. We may have to leave a ‘forwarding address’ on this page identifying the new location of the record. And to ensure that space is always available for this forwarding address, we would have to allocate some minimum space for each record, regardless of its length.

- A record may grow so large that it no longer fits on any one page. We have to deal with this condition by breaking a record into smaller records. The smaller
Large records in real systems: In Sybase ASE, a record can be at most 1962 bytes. This limit is set by the 2 KB log page size, since records are not allowed to be larger than a page. The exceptions to this rule are BLOBs and CLOBs, which consist of a set of bidirectionally linked pages. IBM DB2 and Microsoft SQL Server also do not allow records to span pages, although large objects are allowed to span pages and are handled separately from other data types. In DB2, record size is limited only by the page size; in SQL Server, a record can be at most 8 KB, excluding LOBs. Informix and Oracle 8 allow records to span pages. Informix allows records to be at most 32 KB, while Oracle has no maximum record size; large records are organized as a singly directed list.

records could be chained together—part of each smaller record is a pointer to the next record in the chain—to enable retrieval of the entire original record.

7.8 POINTS TO REVIEW

- Memory in a computer system is arranged into primary storage (cache and main memory), secondary storage (magnetic disks), and tertiary storage (optical disks and tapes). Storage devices that store data persistently are called nonvolatile. (Section 7.1)

- Disks provide inexpensive, nonvolatile storage. The unit of transfer from disk into main memory is called a block or page. Blocks are arranged on tracks on several platters. The time to access a page depends on its location on disk. The access time has three components: the time to move the disk arm to the desired track (seek time), the time to wait for the desired block to rotate under the disk head (rotational delay), and the time to transfer the data (transfer time). (Section 7.1.1)

- Careful placement of pages on the disk to exploit the geometry of a disk can minimize the seek time and rotational delay when pages are read sequentially. (Section 7.1.2)

- A disk array is an arrangement of several disks that are attached to a computer. Performance of a disk array can be increased through data striping and reliability can be increased through redundancy. Different RAID organizations called RAID levels represent different trade-offs between reliability and performance. (Section 7.2)

- In a DBMS, the disk space manager manages space on disk by keeping track of free and used disk blocks. It also provides the abstraction of the data being a collection of disk pages. DBMSs rarely use OS files for performance, functionality, and portability reasons. (Section 7.3)
In a DBMS, all page requests are centrally processed by the buffer manager. The buffer manager transfers pages between the disk and a special area of main memory called the buffer pool, which is divided into page-sized chunks called frames. For each page in the buffer pool, the buffer manager maintains a pin_count, which indicates the number of users of the current page, and a dirty flag, which indicates whether the page has been modified. A requested page is kept in the buffer pool until it is released (unpinned) by all users. Subsequently, a page is written back to disk (if it has been modified while in the buffer pool) when the frame containing it is chosen for replacement. (Section 7.4)

The choice of frame to replace is based on the buffer manager’s replacement policy, for example LRU or clock. Repeated scans of a file can cause sequential flooding if LRU is used. (Section 7.4.1)

A DBMS buffer manager can often predict the access pattern for disk pages. It takes advantage of such opportunities by issuing requests to the disk to prefetch several pages at a time. This technique minimizes disk arm movement and reduces I/O time. A DBMS also needs to be able to force a page to disk to ensure crash recovery. (Section 7.4.2)

Database pages are organized into files, and higher-level DBMS code views the data as a collection of records. (Section 7.5)

The simplest file structure is a heap file, which is an unordered collection of records. Heap files are either organized as a linked list of data pages or as a list of directory pages that refer to the actual pages with data. (Section 7.5.1)

Indexes are auxiliary structures that support efficient retrieval of records based on the values of a search key. (Section 7.5.2)

A page contains a collection of slots, each of which identifies a record. Slotted pages allow a record to be moved around on a page without altering the record identifier or rid, a ⟨page id, slot number⟩ pair. Efficient page organizations exist for either fixed-length records (bitmap of free slots) or variable-length records (slot directory). (Section 7.6)

For fixed-length records, the fields can be stored consecutively and the address of a field can be easily calculated. Variable-length records can be stored with an array of offsets at the beginning of the record or the individual can be fields separated by a delimiter symbol. The organization with an array of offsets offers direct access to fields (which can be important if records are long and contain many fields) and support for null values. (Section 7.7)
EXERCISES

Exercise 7.1 What is the most important difference between a disk and a tape?

Exercise 7.2 Explain the terms seek time, rotational delay, and transfer time.

Exercise 7.3 Both disks and main memory support direct access to any desired location (page). On average, main memory accesses are faster, of course. What is the other important difference (from the perspective of the time required to access a desired page)?

Exercise 7.4 If you have a large file that is frequently scanned sequentially, explain how you would store the pages in the file on a disk.

Exercise 7.5 Consider a disk with a sector size of 512 bytes, 2,000 tracks per surface, 50 sectors per track, 5 double-sided platters, average seek time of 10 msec.

1. What is the capacity of a track in bytes? What is the capacity of each surface? What is the capacity of the disk?
2. How many cylinders does the disk have?
3. Give examples of valid block sizes. Is 256 bytes a valid block size? 2,048? 51,200?
4. If the disk platters rotate at 5,400 rpm (revolutions per minute), what is the maximum rotational delay?
5. Assuming that one track of data can be transferred per revolution, what is the transfer rate?

Exercise 7.6 Consider again the disk specifications from Exercise 7.5 and suppose that a block size of 1,024 bytes is chosen. Suppose that a file containing 100,000 records of 100 bytes each is to be stored on such a disk and that no record is allowed to span two blocks.

1. How many records fit onto a block?
2. How many blocks are required to store the entire file? If the file is arranged sequentially on disk, how many surfaces are needed?
3. How many records of 100 bytes each can be stored using this disk?
4. If pages are stored sequentially on disk, with page 1 on block 1 of track 1, what is the page stored on block 1 of track 1 on the next disk surface? How would your answer change if the disk were capable of reading/writing from all heads in parallel?
5. What is the time required to read a file containing 100,000 records of 100 bytes each sequentially? Again, how would your answer change if the disk were capable of reading/writing from all heads in parallel (and the data was arranged optimally)?
6. What is the time required to read a file containing 100,000 records of 100 bytes each in some random order? Note that in order to read a record, the block containing the record has to be fetched from disk. Assume that each block request incurs the average seek time and rotational delay.

Exercise 7.7 Explain what the buffer manager must do to process a read request for a page. What happens if the requested page is in the pool but not pinned?
Exercise 7.8 When does a buffer manager write a page to disk?

Exercise 7.9 What does it mean to say that a page is pinned in the buffer pool? Who is responsible for pinning pages? Who is responsible for unpinning pages?

Exercise 7.10 When a page in the buffer pool is modified, how does the DBMS ensure that this change is propagated to disk? (Explain the role of the buffer manager as well as the modifier of the page.)

Exercise 7.11 What happens if there is a page request when all pages in the buffer pool are dirty?

Exercise 7.12 What is sequential flooding of the buffer pool?

Exercise 7.13 Name an important capability of a DBMS buffer manager that is not supported by a typical operating system’s buffer manager.

Exercise 7.14 Explain the term prefetching. Why is it important?

Exercise 7.15 Modern disks often have their own main memory caches, typically about one MB, and use this to do prefetching of pages. The rationale for this technique is the empirical observation that if a disk page is requested by some (not necessarily database!) application, 80 percent of the time the next page is requested as well. So the disk gambles by reading ahead.

1. Give a nontechnical reason that a DBMS may not want to rely on prefetching controlled by the disk.
2. Explain the impact on the disk’s cache of several queries running concurrently, each scanning a different file.
3. Can the above problem be addressed by the DBMS buffer manager doing its own prefetching? Explain.
4. Modern disks support segmented caches, with about four to six segments, each of which is used to cache pages from a different file. Does this technique help, with respect to the above problem? Given this technique, does it matter whether the DBMS buffer manager also does prefetching?

Exercise 7.16 Describe two possible record formats. What are the trade-offs between them?

Exercise 7.17 Describe two possible page formats. What are the trade-offs between them?

Exercise 7.18 Consider the page format for variable-length records that uses a slot directory.

1. One approach to managing the slot directory is to use a maximum size (i.e., a maximum number of slots) and to allocate the directory array when the page is created. Discuss the pros and cons of this approach with respect to the approach discussed in the text.
2. Suggest a modification to this page format that would allow us to sort records (according to the value in some field) without moving records and without changing the record ids.
Exercise 7.19 Consider the two internal organizations for heap files (using lists of pages and a directory of pages) discussed in the text.

1. Describe them briefly and explain the trade-offs. Which organization would you choose if records are variable in length?
2. Can you suggest a single page format to implement both internal file organizations?

Exercise 7.20 Consider a list-based organization of the pages in a heap file in which two lists are maintained: a list of all pages in the file and a list of all pages with free space. In contrast, the list-based organization discussed in the text maintains a list of full pages and a list of pages with free space.

1. What are the trade-offs, if any? Is one of them clearly superior?
2. For each of these organizations, describe a page format that can be used to implement it.

Exercise 7.21 Modern disk drives store more sectors on the outer tracks than the inner tracks. Since the rotation speed is constant, the sequential data transfer rate is also higher on the outer tracks. The seek time and rotational delay are unchanged. Considering this information, explain good strategies for placing files with the following kinds of access patterns:

1. Frequent, random accesses to a small file (e.g., catalog relations).
2. Sequential scans of a large file (e.g., selection from a relation with no index).
3. Random accesses to a large file via an index (e.g., selection from a relation via the index).

PROJECT-BASED EXERCISES

Exercise 7.22 Study the public interfaces for the disk space manager, the buffer manager, and the heap file layer in Minibase.

1. Are heap files with variable-length records supported?
2. What page format is used in Minibase heap files?
3. What happens if you insert a record whose length is greater than the page size?
4. How is free space handled in Minibase?
5. Note to Instructors: See Appendix B for additional project-based exercises.

BIBLIOGRAPHIC NOTES

Salzberg [564] and Wiederhold [681] discuss secondary storage devices and file organizations in detail.
RAID was originally proposed by Patterson, Gibson, and Katz [512]. The article by Chen et al. provides an excellent survey of RAID [144]. Books about RAID include Gibson’s dissertation [269] and the publications from the RAID Advisory Board [527].

The design and implementation of storage managers is discussed in [54, 113, 413, 629, 184]. With the exception of [184], these systems emphasize extensibility, and the papers contain much of interest from that standpoint as well. Other papers that cover storage management issues in the context of significant implemented prototype systems are [415] and [513]. The Dali storage manager, which is optimized for main memory databases, is described in [345]. Three techniques for implementing long fields are compared in [83].

Stonebraker discusses operating systems issues in the context of databases in [626]. Several buffer management policies for database systems are compared in [150]. Buffer management is also studied in [101, 142, 223, 198].
8 FILE ORGANIZATIONS & INDEXES

If you don’t find it in the index, look very carefully through the entire catalog.

—Sears, Roebuck, and Co., Consumers’ Guide, 1897

A file organization is a way of arranging the records in a file when the file is stored on disk. A file of records is likely to be accessed and modified in a variety of ways, and different ways of arranging the records enable different operations over the file to be carried out efficiently. For example, if we want to retrieve employee records in alphabetical order, sorting the file by name is a good file organization. On the other hand, if we want to retrieve all employees whose salary is in a given range, sorting employee records by name is not a good file organization. A DBMS supports several file organization techniques, and an important task of a DBA is to choose a good organization for each file, based on its expected pattern of use.

We begin this chapter with a discussion in Section 8.1 of the cost model that we use in this book. In Section 8.2, we present a simplified analysis of three basic file organizations: files of randomly ordered records (i.e., heap files), files sorted on some field, and files that are hashed on some fields. Our objective is to emphasize the importance of choosing an appropriate file organization.

Each file organization makes certain operations efficient, but often we are interested in supporting more than one operation. For example, sorting a file of employee records on the name field makes it easy to retrieve employees in alphabetical order, but we may also want to retrieve all employees who are 55 years old; for this, we would have to scan the entire file. To deal with such situations, a DBMS builds an index, as we described in Section 7.5.2. An index on a file is designed to speed up operations that are not efficiently supported by the basic organization of records in that file. Later chapters cover several specific index data structures; in this chapter we focus on properties of indexes that do not depend on the specific index data structure used.

Section 8.3 introduces indexing as a general technique that can speed up retrieval of records with given values in the search field. Section 8.4 discusses some important properties of indexes, and Section 8.5 discusses DBMS commands to create an index.
8.1 COST MODEL

In this section we introduce a cost model that allows us to estimate the cost (in terms of execution time) of different database operations. We will use the following notation and assumptions in our analysis. There are $B$ data pages with $R$ records per page. The average time to read or write a disk page is $D$, and the average time to process a record (e.g., to compare a field value to a selection constant) is $C$. In the hashed file organization, we will use a function, called a hash function, to map a record into a range of numbers; the time required to apply the hash function to a record is $H$.

Typical values today are $D = 15$ milliseconds, $C$ and $H = 100$ nanoseconds; we therefore expect the cost of I/O to dominate. This conclusion is supported by current hardware trends, in which CPU speeds are steadily rising, whereas disk speeds are not increasing at a similar pace. On the other hand, as main memory sizes increase, a much larger fraction of the needed pages are likely to fit in memory, leading to fewer I/O requests.

We therefore use the number of disk page I/Os as our cost metric in this book.

- We emphasize that real systems must consider other aspects of cost, such as CPU costs (and transmission costs in a distributed database). However, our goal is primarily to present the underlying algorithms and to illustrate how costs can be estimated. Therefore, for simplicity, we have chosen to concentrate on only the I/O component of cost. Given the fact that I/O is often (even typically) the dominant component of the cost of database operations, considering I/O costs gives us a good first approximation to the true costs.

- Even with our decision to focus on I/O costs, an accurate model would be too complex for our purposes of conveying the essential ideas in a simple way. We have therefore chosen to use a simplistic model in which we just count the number of pages that are read from or written to disk as a measure of I/O. We have ignored the important issue of blocked access—typically, disk systems allow us to read a block of contiguous pages in a single I/O request. The cost is equal to the time required to seek the first page in the block and to transfer all pages in the block. Such blocked access can be much cheaper than issuing one I/O request per page in the block, especially if these requests do not follow consecutively: We would have an additional seek cost for each page in the block.

This discussion of the cost metric we have chosen must be kept in mind when we discuss the cost of various algorithms in this chapter and in later chapters. We discuss the implications of the cost model whenever our simplifying assumptions are likely to affect the conclusions drawn from our analysis in an important way.
8.2 COMPARISON OF THREE FILE ORGANIZATIONS

We now compare the costs of some simple operations for three basic file organizations: files of randomly ordered records, or heap files; files sorted on a sequence of fields; and files that are hashed on a sequence of fields. For sorted and hashed files, the sequence of fields (e.g., salary, age) on which the file is sorted or hashed is called the search key. Note that the search key for an index can be any sequence of one or more fields; it need not uniquely identify records. We observe that there is an unfortunate overloading of the term key in the database literature. A primary key or candidate key (fields that uniquely identify a record; see Chapter 3) is unrelated to the concept of a search key.

Our goal is to emphasize how important the choice of an appropriate file organization can be. The operations that we consider are described below.

- **Scan:** Fetch all records in the file. The pages in the file must be fetched from disk into the buffer pool. There is also a CPU overhead per record for locating the record on the page (in the pool).

- **Search with equality selection:** Fetch all records that satisfy an equality selection, for example, “Find the Students record for the student with sid 23.” Pages that contain qualifying records must be fetched from disk, and qualifying records must be located within retrieved pages.

- **Search with range selection:** Fetch all records that satisfy a range selection, for example, “Find all Students records with name alphabetically after ‘Smith.’”

- **Insert:** Insert a given record into the file. We must identify the page in the file into which the new record must be inserted, fetch that page from disk, modify it to include the new record, and then write back the modified page. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.

- **Delete:** Delete a record that is specified using its rid. We must identify the page that contains the record, fetch it from disk, modify it, and write it back. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.

8.2.1 Heap Files

**Scan:** The cost is \(B(D + RC)\) because we must retrieve each of \(B\) pages taking time \(D\) per page, and for each page, process \(R\) records taking time \(C\) per record.

**Search with equality selection:** Suppose that we know in advance that exactly one record matches the desired equality selection, that is, the selection is specified on a candidate key. On average, we must scan half the file, assuming that the record exists.
and the distribution of values in the search field is uniform. For each retrieved data page, we must check all records on the page to see if it is the desired record. The cost is $0.5B(D + RC)$. If there is no record that satisfies the selection, however, we must scan the entire file to verify this.

If the selection is not on a candidate key field (e.g., “Find students aged 18”), we always have to scan the entire file because several records with $age = 18$ could be dispersed all over the file, and we have no idea how many such records exist.

**Search with range selection:** The entire file must be scanned because qualifying records could appear anywhere in the file, and we do not know how many qualifying records exist. The cost is $B(D + RC)$.

**Insert:** We assume that records are always inserted at the end of the file. We must fetch the last page in the file, add the record, and write the page back. The cost is $2D + C$.

**Delete:** We must find the record, remove the record from the page, and write the modified page back. We assume that no attempt is made to compact the file to reclaim the free space created by deletions, for simplicity.\(^1\) The cost is the cost of searching plus $C + D$.

We assume that the record to be deleted is specified using the record id. Since the page id can easily be obtained from the record id, we can directly read in the page. The cost of searching is therefore $D$.

If the record to be deleted is specified using an equality or range condition on some fields, the cost of searching is given in our discussion of equality and range selections. The cost of deletion is also affected by the number of qualifying records, since all pages containing such records must be modified.

### 8.2.2 Sorted Files

**Scan:** The cost is $B(D + RC)$ because all pages must be examined. Note that this case is no better or worse than the case of unordered files. However, the order in which records are retrieved corresponds to the sort order.

**Search with equality selection:** We assume that the equality selection is specified on the field by which the file is sorted; if not, the cost is identical to that for a heap

---

\(^1\)In practice, a directory or other data structure is used to keep track of free space, and records are inserted into the first available free slot, as discussed in Chapter 7. This increases the cost of insertion and deletion a little, but not enough to affect our comparison of heap files, sorted files, and hashed files.
file. We can locate the first page containing the desired record or records, should any qualifying records exist, with a binary search in $\log_2 B$ steps. (This analysis assumes that the pages in the sorted file are stored sequentially, and we can retrieve the $i$th page on the file directly in one disk I/O. This assumption is not valid if, for example, the sorted file is implemented as a heap file using the linked-list organization, with pages in the appropriate sorted order.) Each step requires a disk I/O and two comparisons. Once the page is known, the first qualifying record can again be located by a binary search of the page at a cost of $C\log_2 R$. The cost is $D\log_2 B + C\log_2 R$, which is a significant improvement over searching heap files.

If there are several qualifying records (e.g., “Find all students aged 18”), they are guaranteed to be adjacent to each other due to the sorting on age, and so the cost of retrieving all such records is the cost of locating the first such record ($D\log_2 B + C\log_2 R$) plus the cost of reading all the qualifying records in sequential order. Typically, all qualifying records fit on a single page. If there are no qualifying records, this is established by the search for the first qualifying record, which finds the page that would have contained a qualifying record, had one existed, and searches that page.

**Search with range selection:** Again assuming that the range selection is on the sort field, the first record that satisfies the selection is located as it is for search with equality. Subsequently, data pages are sequentially retrieved until a record is found that does not satisfy the range selection; this is similar to an equality search with many qualifying records.

The cost is the cost of search plus the cost of retrieving the set of records that satisfy the search. The cost of the search includes the cost of fetching the first page containing qualifying, or matching, records. For small range selections, all qualifying records appear on this page. For larger range selections, we have to fetch additional pages containing matching records.

**Insert:** To insert a record while preserving the sort order, we must first find the correct position in the file, add the record, and then fetch and rewrite all subsequent pages (because all the old records will be shifted by one slot, assuming that the file has no empty slots). On average, we can assume that the inserted record belongs in the middle of the file. Thus, we must read the latter half of the file and then write it back after adding the new record. The cost is therefore the cost of searching to find the position of the new record plus $2 \times (0.5B(D + RC))$, that is, search cost plus $B(D + RC)$.

**Delete:** We must search for the record, remove the record from the page, and write the modified page back. We must also read and write all subsequent pages because all
records that follow the deleted record must be moved up to compact the free space.\footnote{Unlike a heap file, there is no inexpensive way to manage free space, so we account for the cost of compacting a file when a record is deleted.} The cost is the same as for an insert, that is, search cost plus $B(D + RC)$. Given the rid of the record to delete, we can fetch the page containing the record directly.

If records to be deleted are specified by an equality or range condition, the cost of deletion depends on the number of qualifying records. If the condition is specified on the sort field, qualifying records are guaranteed to be contiguous due to the sorting, and the first qualifying record can be located using binary search.

### 8.2.3 Hashed Files

A simple hashed file organization enables us to locate records with a given search key value quickly, for example, “Find the Students record for Joe,” if the file is hashed on the name field.

The pages in a hashed file are grouped into buckets. Given a bucket number, the hashed file structure allows us to find the primary page for that bucket. The bucket to which a record belongs can be determined by applying a special function called a hash function, to the search field(s). On inserts, a record is inserted into the appropriate bucket, with additional ‘overflow’ pages allocated if the primary page for the bucket becomes full. The overflow pages for each bucket are maintained in a linked list. To search for a record with a given search key value, we simply apply the hash function to identify the bucket to which such records belong and look at all pages in that bucket.

This organization is called a static hashed file, and its main drawback is that long chains of overflow pages can develop. This can affect performance because all pages in a bucket have to be searched. Dynamic hash structures that address this problem are known, and we discuss them in Chapter 10; for the analysis in this chapter, we will simply assume that there are no overflow pages.

**Scan:** In a hashed file, pages are kept at about 80 percent occupancy (to leave some space for future insertions and minimize overflow pages as the file expands). This is achieved by adding a new page to a bucket when each existing page is 80 percent full, when records are initially organized into a hashed file structure. Thus, the number of pages, and the cost of scanning all the data pages, is about 1.25 times the cost of scanning an unordered file, that is, $1.25B(D + RC)$.

**Search with equality selection:** This operation is supported very efficiently if the selection is on the search key for the hashed file. (Otherwise, the entire file must
be scanned.) The cost of identifying the page that contains qualifying records is $H$; assuming that this bucket consists of just one page (i.e., no overflow pages), retrieving it costs $D$. The cost is $H + D + 0.5RC$ if we assume that we find the record after scanning half the records on the page. This is even lower than the cost for sorted files. If there are several qualifying records, or none, we still have to retrieve just one page, but we must scan the entire page.

Note that the hash function associated with a hashed file maps a record to a bucket based on the values in all the search key fields; if the value for any one of these fields is not specified, we cannot tell which bucket the record belongs to. Thus, if the selection is not an equality condition on all the search key fields, we have to scan the entire file.

**Search with range selection:** The hash structure offers no help; even if the range selection is on the search key, the entire file must be scanned. The cost is $1.25B(D + RC)$.

**Insert:** The appropriate page must be located, modified, and then written back. The cost is the cost of search plus $C + D$.

**Delete:** We must search for the record, remove it from the page, and write the modified page back. The cost is again the cost of search plus $C + D$ (for writing the modified page).

If records to be deleted are specified using an equality condition on the search key, all qualifying records are guaranteed to be in the same bucket, which can be identified by applying the hash function.

### 8.2.4 Choosing a File Organization

Figure 8.1 compares I/O costs for the three file organizations. A heap file has good storage efficiency and supports fast scan, insertion, and deletion of records. However, it is slow for searches.

<table>
<thead>
<tr>
<th>File Type</th>
<th>Scan: $BD$</th>
<th>Equality Search: $0.5BD$</th>
<th>Range Search: $BD$</th>
<th>Insert: $2D$</th>
<th>Delete: $Search + D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted</td>
<td>$BD$</td>
<td>$D$log$_{2}B$</td>
<td>$D$log$_{2}B$+# matches</td>
<td>$Search + BD$</td>
<td>$Search + BD$</td>
</tr>
<tr>
<td>Hashed</td>
<td>$1.25BD$</td>
<td>$D$</td>
<td>$1.25BD$</td>
<td>$2D$</td>
<td>$Search + D$</td>
</tr>
</tbody>
</table>

*Figure 8.1  A Comparison of I/O Costs*
A sorted file also offers good storage efficiency, but insertion and deletion of records is slow. It is quite fast for searches, and it is the best structure for range selections. It is worth noting that in a real DBMS, a file is almost never kept fully sorted. A structure called a B+ tree, which we will discuss in Chapter 9, offers all the advantages of a sorted file and supports inserts and deletes efficiently. (There is a space overhead for these benefits, relative to a sorted file, but the trade-off is well worth it.)

Files are sometimes kept ‘almost sorted’ in that they are originally sorted, with some free space left on each page to accommodate future insertions, but once this space is used, overflow pages are used to handle insertions. The cost of insertion and deletion is similar to a heap file, but the degree of sorting deteriorates as the file grows.

A hashed file does not utilize space quite as well as a sorted file, but insertions and deletions are fast, and equality selections are very fast. However, the structure offers no support for range selections, and full file scans are a little slower; the lower space utilization means that files contain more pages.

In summary, Figure 8.1 demonstrates that no one file organization is uniformly superior in all situations. An unordered file is best if only full file scans are desired. A hashed file is best if the most common operation is an equality selection. A sorted file is best if range selections are desired. The organizations that we have studied here can be improved on—the problems of overflow pages in static hashing can be overcome by using dynamic hashing structures, and the high cost of inserts and deletes in a sorted file can be overcome by using tree-structured indexes—but the main observation, that the choice of an appropriate file organization depends on how the file is commonly used, remains valid.

### 8.3 OVERVIEW OF INDEXES

As we noted earlier, an index on a file is an auxiliary structure designed to speed up operations that are not efficiently supported by the basic organization of records in that file.

An index can be viewed as a collection of data entries, with an efficient way to locate all data entries with search key value $k$. Each such data entry, which we denote as $k^*$, contains enough information to enable us to retrieve (one or more) data records with search key value $k$. (Note that a data entry is, in general, different from a data record!) Figure 8.2 shows an index with search key $sal$ that contains $(sal, rid)$ pairs as data entries. The $rid$ component of a data entry in this index is a pointer to a record with search key value $sal$.

Two important questions to consider are:
How are data entries organized in order to support efficient retrieval of data entries with a given search key value?

Exactly what is stored as a data entry?

One way to organize data entries is to hash data entries on the search key. In this approach, we essentially treat the collection of data entries as a file of records, hashed on the search key. This is how the index on sal shown in Figure 8.2 is organized. The hash function \( h \) for this example is quite simple; it converts the search key value to its binary representation and uses the two least significant bits as the bucket identifier. Another way to organize data entries is to build a data structure that directs a search for data entries. Several index data structures are known that allow us to efficiently find data entries with a given search key value. We will study tree-based index structures in Chapter 9 and hash-based index structures in Chapter 10.

We consider what is stored in a data entry in the following section.

### 8.3.1 Alternatives for Data Entries in an Index

A data entry \( k^* \) allows us to retrieve one or more data records with key value \( k \). We need to consider three main alternatives:

1. A data entry \( k^* \) is an actual data record (with search key value \( k \)).
2. A data entry is a \( \langle k, rid \rangle \) pair, where \( rid \) is the record id of a data record with search key value \( k \).
3. A data entry is a \( \langle k, rid-list \rangle \) pair, where \( rid-list \) is a list of record ids of data records with search key value \( k \).
Observe that if an index uses Alternative (1), there is no need to store the data records separately, in addition to the contents of the index. We can think of such an index as a special file organization that can be used instead of a sorted file or a heap file organization. Figure 8.2 illustrates Alternatives (1) and (2). The file of employee records is hashed on age; we can think of this as an index structure in which a hash function is applied to the age value to locate the bucket for a record and Alternative (1) is used for data entries. The index on sal also uses hashing to locate data entries, which are now \( \langle \text{sal}, \text{rid of employee record} \rangle \) pairs; that is, Alternative (2) is used for data entries.

Alternatives (2) and (3), which contain data entries that point to data records, are independent of the file organization that is used for the indexed file (i.e., the file that contains the data records). Alternative (3) offers better space utilization than Alternative (2), but data entries are variable in length, depending on the number of data records with a given search key value.

If we want to build more than one index on a collection of data records, for example, we want to build indexes on both the age and the sal fields as illustrated in Figure 8.2, at most one of the indexes should use Alternative (1) because we want to avoid storing data records multiple times.

We note that different index data structures used to speed up searches for data entries with a given search key can be combined with any of the three alternatives for data entries.

### 8.4 PROPERTIES OF INDEXES

In this section, we discuss some important properties of an index that affect the efficiency of searches using the index.

#### 8.4.1 Clustered versus Unclustered Indexes

When a file is organized so that the ordering of data records is the same as or close to the ordering of data entries in some index, we say that the index is **clustered**. An index that uses Alternative (1) is clustered, by definition. An index that uses Alternative (2) or Alternative (3) can be a clustered index only if the data records are sorted on the search key field. Otherwise, the order of the data records is random, defined purely by their physical order, and there is no reasonable way to arrange the data entries in the index in the same order. (Indexes based on hashing do not store data entries in sorted order by search key, so a hash index is clustered only if it uses Alternative (1).)
Indexes that maintain data entries in sorted order by search key use a collection of *index entries*, organized into a tree structure, to guide searches for data entries, which are stored at the leaf level of the tree in sorted order. Clustered and unclustered tree indexes are illustrated in Figures 8.3 and 8.4; we discuss tree-structured indexes further in Chapter 9. For simplicity, in Figure 8.3 we assume that the underlying file of data records is fully sorted.

![Clustered Tree Index Using Alternative (2)](image1)

**Figure 8.3** Clustered Tree Index Using Alternative (2)

![Unclustered Tree Index Using Alternative (2)](image2)

**Figure 8.4** Unclustered Tree Index Using Alternative (2)

In practice, data records are rarely maintained in fully sorted order, unless data records are stored in an index using Alternative (1), because of the high overhead of moving data records around to preserve the sort order as records are inserted and deleted. Typically, the records are sorted initially and each page is left with some free space to absorb future insertions. If the free space on a page is subsequently used up (by records...
inserted after the initial sorting step), further insertions to this page are handled using a linked list of overflow pages. Thus, after a while, the order of records only approximates the intended sorted order, and the file must be reorganized (i.e., sorted afresh) to ensure good performance.

Thus, clustered indexes are relatively expensive to maintain when the file is updated. Another reason clustered indexes are expensive to maintain is that data entries may have to be moved across pages, and if records are identified by a combination of page id and slot, as is often the case, all places in the database that point to a moved record (typically, entries in other indexes for the same collection of records) must also be updated to point to the new location; these additional updates can be very time-consuming.

A data file can be clustered on at most one search key, which means that we can have at most one clustered index on a data file. An index that is not clustered is called an unclustered index; we can have several unclustered indexes on a data file. Suppose that Students records are sorted by age; an index on age that stores data entries in sorted order by age is a clustered index. If in addition we have an index on the gpa field, the latter must be an unclustered index.

The cost of using an index to answer a range search query can vary tremendously based on whether the index is clustered. If the index is clustered, the rids in qualifying data entries point to a contiguous collection of records, as Figure 8.3 illustrates, and we need to retrieve only a few data pages. If the index is unclustered, each qualifying data entry could contain a rid that points to a distinct data page, leading to as many data page I/Os as the number of data entries that match the range selection! This point is discussed further in Chapters 11 and 16.

8.4.2 Dense versus Sparse Indexes

An index is said to be dense if it contains (at least) one data entry for every search key value that appears in a record in the indexed file.3 A sparse index contains one entry for each page of records in the data file. Alternative (1) for data entries always leads to a dense index. Alternative (2) can be used to build either dense or sparse indexes. Alternative (3) is typically only used to build a dense index.

We illustrate sparse and dense indexes in Figure 8.5. A data file of records with three fields (name, age, and sal) is shown with two simple indexes on it, both of which use Alternative (2) for data entry format. The first index is a sparse, clustered index on name. Notice how the order of data entries in the index corresponds to the order of

---

3We say ‘at least’ because several data entries could have the same search key value if there are duplicates and we use Alternative (2).
records in the data file. There is one data entry per page of data records. The second index is a dense, unclustered index on the age field. Notice that the order of data entries in the index differs from the order of data records. There is one data entry in the index per record in the data file (because we use Alternative (2)).

![Figure 8.5 Sparse versus Dense Indexes](image)

We cannot build a sparse index that is not clustered. Thus, we can have at most one sparse index. A sparse index is typically much smaller than a dense index. On the other hand, some very useful optimization techniques rely on an index being dense (Chapter 16).

A data file is said to be inverted on a field if there is a dense secondary index on this field. A fully inverted file is one in which there is a dense secondary index on each field that does not appear in the primary key.\(^4\)

### 8.4.3 Primary and Secondary Indexes

An index on a set of fields that includes the primary key is called a primary index. An index that is not a primary index is called a secondary index. (The terms primary index and secondary index are sometimes used with a different meaning: An index that uses Alternative (1) is called a primary index, and one that uses Alternatives (2) or (3) is called a secondary index. We will be consistent with the definitions presented earlier, but the reader should be aware of this lack of standard terminology in the literature.)

---

\(^4\)This terminology arises from the observation that these index structures allow us to take the value in a non key field and get the values in key fields, which is the inverse of the more intuitive case in which we use the values of the key fields to locate the record.
Two data entries are said to be duplicates if they have the same value for the search key field associated with the index. A primary index is guaranteed not to contain duplicates, but an index on other (collections of) fields can contain duplicates. Thus, in general, a secondary index contains duplicates. If we know that no duplicates exist, that is, we know that the search key contains some candidate key, we call the index a unique index.

### 8.4.4 Indexes Using Composite Search Keys

The search key for an index can contain several fields; such keys are called composite search keys or concatenated keys. As an example, consider a collection of employee records, with fields name, age, and sal, stored in sorted order by name. Figure 8.6 illustrates the difference between a composite index with key \(\langle \text{age}, \text{sal} \rangle\), a composite index with key \(\langle \text{sal}, \text{age} \rangle\), an index with key age, and an index with key sal. All indexes shown in the figure use Alternative (2) for data entries.

If the search key is composite, an equality query is one in which each field in the search key is bound to a constant. For example, we can ask to retrieve all data entries with age = 20 and sal = 10. The hashed file organization supports only equality queries, since a hash function identifies the bucket containing desired records only if a value is specified for each field in the search key.

A range query is one in which not all fields in the search key are bound to constants. For example, we can ask to retrieve all data entries with age = 20; this query implies that any value is acceptable for the sal field. As another example of a range query, we can ask to retrieve all data entries with age < 30 and sal > 40.
8.5 INDEX SPECIFICATION IN SQL-92

The SQL-92 standard does not include any statement for creating or dropping index structures. In fact, the standard does not even require SQL implementations to support indexes! In practice, of course, every commercial relational DBMS supports one or more kinds of indexes. The following command to create a B+ tree index—we discuss B+ tree indexes in Chapter 9—is illustrative:

```
CREATE INDEX IndAgeRating ON Students
    WITH STRUCTURE = BTREE,
    KEY = (age, gpa)
```

This specifies that a B+ tree index is to be created on the Students table using the concatenation of the `age` and `gpa` columns as the key. Thus, key values are pairs of the form `(age, gpa)`, and there is a distinct entry for each such pair. Once the index is created, it is automatically maintained by the DBMS adding/removing data entries in response to inserts/deletes of records on the Students relation.

8.6 POINTS TO REVIEW

- A file organization is a way of arranging records in a file. In our discussion of different file organizations, we use a simple cost model that uses the number of disk page I/Os as the cost metric. (Section 8.1)

- We compare three basic file organizations (heap files, sorted files, and hashed files) using the following operations: scan, equality search, range search, insert, and delete. The choice of file organization can have a significant impact on performance. (Section 8.2)

- An index is a data structure that speeds up certain operations on a file. The operations involve a search key, which is a set of record fields (in most cases a single field). The elements of an index are called data entries. Data entries can be actual data records, `(search-key, rid)` pairs, or `(search-key, rid-list)` pairs. A given file of data records can have several indexes, each with a different search key. (Section 8.3)

- In a clustered index, the order of records in the file matches the order of data entries in the index. An index is called dense if there is at least one data entry per search key that appears in the file; otherwise the index is called sparse. An index is called a primary index if the search key includes the primary key; otherwise it is called a secondary index. If a search key contains several fields it is called a composite key. (Section 8.4)

- SQL-92 does not include statements for management of index structures, and so there some variation in index-related commands across different DBMSs. (Section 8.5)
EXERCISES

Exercise 8.1 What are the main conclusions that you can draw from the discussion of the three file organizations?

Exercise 8.2 Consider a delete specified using an equality condition. What is the cost if no record qualifies? What is the cost if the condition is not on a key?

Exercise 8.3 Which of the three basic file organizations would you choose for a file where the most frequent operations are as follows?

1. Search for records based on a range of field values.
2. Perform inserts and scans where the order of records does not matter.
3. Search for a record based on a particular field value.

Exercise 8.4 Explain the difference between each of the following:

1. Primary versus secondary indexes.
2. Dense versus sparse indexes.
3. Clustered versus unclustered indexes.

If you were about to create an index on a relation, what considerations would guide your choice with respect to each pair of properties listed above?

Exercise 8.5 Consider a relation stored as a randomly ordered file for which the only index is an unclustered index on a field called $sal$. If you want to retrieve all records with $sal > 20$, is using the index always the best alternative? Explain.

Exercise 8.6 If an index contains data records as ‘data entries’, is it clustered or unclustered? Dense or sparse?

Exercise 8.7 Consider Alternatives (1), (2) and (3) for ‘data entries’ in an index, as discussed in Section 8.3.1. Are they all suitable for secondary indexes? Explain.

Exercise 8.8 Consider the instance of the Students relation shown in Figure 8.7, sorted by $age$: For the purposes of this question, assume that these tuples are stored in a sorted file in the order shown; the first tuple is in page 1, slot 1; the second tuple is in page 1, slot 2; and so on. Each page can store up to three data records. You can use $\langle page-id, slot \rangle$ to identify a tuple.

List the data entries in each of the following indexes. If the order of entries is significant, say so and explain why. If such an index cannot be constructed, say so and explain why.

1. A dense index on $age$ using Alternative (1).
2. A dense index on $age$ using Alternative (2).
3. A dense index on $age$ using Alternative (3).
4. A sparse index on $age$ using Alternative (1).
5. A sparse index on age using Alternative (2).
6. A sparse index on age using Alternative (3).
7. A dense index on gpa using Alternative (1).
8. A dense index on gpa using Alternative (2).
9. A dense index on gpa using Alternative (3).
10. A sparse index on gpa using Alternative (1).
11. A sparse index on gpa using Alternative (2).
12. A sparse index on gpa using Alternative (3).

PROJECT-BASED EXERCISES

Exercise 8.9 Answer the following questions:

1. What indexing techniques are supported in Minibase?
2. What alternatives for data entries are supported?
3. Are clustered indexes supported? Are sparse indexes supported?

BIBLIOGRAPHIC NOTES

Several books discuss file organizations in detail [25, 266, 381, 461, 564, 606, 680].
We now consider two index data structures, called ISAM and B+ trees, based on tree organizations. These structures provide efficient support for range searches, including sorted file scans as a special case. Unlike sorted files, these index structures support efficient insertion and deletion. They also provide support for equality selections, although they are not as efficient in this case as hash-based indexes, which are discussed in Chapter 10.

An ISAM tree is a static index structure that is effective when the file is not frequently updated, but it is unsuitable for files that grow and shrink a lot. We discuss ISAM in Section 9.1. The B+ tree is a dynamic structure that adjusts to changes in the file gracefully. It is the most widely used index structure because it adjusts well to changes and supports both equality and range queries. We introduce B+ trees in Section 9.2. We cover B+ trees in detail in the remaining sections. Section 9.3 describes the format of a tree node. Section 9.4 considers how to search for records by using a B+ tree index. Section 9.5 presents the algorithm for inserting records into a B+ tree, and Section 9.6 presents the deletion algorithm. Section 9.7 discusses how duplicates are handled. We conclude with a discussion of some practical issues concerning B+ trees in Section 9.8.

**Notation:** In the ISAM and B+ tree structures, leaf pages contain data entries, according to the terminology introduced in Chapter 8. For convenience, we will denote a data entry with search key value $k$ as $k^*$. Non-leaf pages contain index entries of the form $(\text{search key value, page id})$ and are used to direct the search for a desired data entry (which is stored in some leaf). We will often simply use entry where the context makes the nature of the entry (index or data) clear.

---

1ISAM stands for Indexed Sequential Access Method.
Chapter 9

9.1 INDEXED SEQUENTIAL ACCESS METHOD (ISAM)

To understand the motivation for the ISAM technique, it is useful to begin with a
simple sorted file. Consider a file of Students records sorted by gpa. To answer a range
selection such as “Find all students with a gpa higher than 3.0,” we must identify the
first such student by doing a binary search of the file and then scan the file from that
point on. If the file is large, the initial binary search can be quite expensive; can we
improve upon this method?

One idea is to create a second file with one record per page in the original (data) file, of
the form (first key on page, pointer to page), again sorted by the key attribute (which
is gpa in our example). The format of a page in the second index file is illustrated in
Figure 9.1.

![Figure 9.1 Format of an Index Page](image)

We refer to pairs of the form (key, pointer) as entries. Notice that each index page
contains one pointer more than the number of keys—each key serves as a separator for
the contents of the pages pointed to by the pointers to its left and right. This structure
is illustrated in Figure 9.2.

![Figure 9.2 One-Level Index Structure](image)

We can do a binary search of the index file to identify the page containing the first
key (gpa) value that satisfies the range selection (in our example, the first student
with gpa over 3.0) and follow the pointer to the page containing the first data record
with that key value. We can then scan the data file sequentially from that point on
to retrieve other qualifying records. This example uses the index to find the first
data page containing a Students record with gpa greater than 3.0, and the data file is
scanned from that point on to retrieve other such Students records.
Because the size of an entry in the index file (key value and page id) is likely to be much smaller than the size of a page, and only one such entry exists per page of the data file, the index file is likely to be much smaller than the data file; thus, a binary search of the index file is much faster than a binary search of the data file. However, a binary search of the index file could still be fairly expensive, and the index file is typically still large enough to make inserts and deletes expensive.

The potential large size of the index file motivates the ISAM idea: Why not apply the previous step of building an auxiliary file on the index file and so on recursively until the final auxiliary file fits on one page? This repeated construction of a one-level index leads to a tree structure that is illustrated in Figure 9.3. The data entries of the ISAM index are in the leaf pages of the tree and additional overflow pages that are chained to some leaf page. In addition, some systems carefully organize the layout of pages so that page boundaries correspond closely to the physical characteristics of the underlying storage device. The ISAM structure is completely static (except for the overflow pages, of which it is hoped, there will be few) and facilitates such low-level optimizations.

Each tree node is a disk page, and all the data resides in the leaf pages. This corresponds to an index that uses Alternative (1) for data entries, in terms of the alternatives described in Chapter 8; we can create an index with Alternative (2) by storing the data records in a separate file and storing \(\langle \text{key}, \text{rid} \rangle\) pairs in the leaf pages of the ISAM index. When the file is created, all leaf pages are allocated sequentially and sorted on the search key value. (If Alternatives (2) or (3) are used, the data records are created and sorted before allocating the leaf pages of the ISAM index.) The non-leaf level pages are then allocated. If there are several inserts to the file subsequently, so that more entries are inserted into a leaf than will fit onto a single page, additional pages are needed because the index structure is static. These additional pages are allocated from an overflow area. The allocation of pages is illustrated in Figure 9.4.
The basic operations of insertion, deletion, and search are all quite straightforward. For an equality selection search, we start at the root node and determine which subtree to search by comparing the value in the search field of the given record with the key values in the node. (The search algorithm is identical to that for a B+ tree; we present this algorithm in more detail later.) For a range query, the starting point in the data (or leaf) level is determined similarly, and data pages are then retrieved sequentially. For inserts and deletes, the appropriate page is determined as for a search, and the record is inserted or deleted with overflow pages added if necessary.

The following example illustrates the ISAM index structure. Consider the tree shown in Figure 9.5. All searches begin at the root. For example, to locate a record with the key value 27, we start at the root and follow the left pointer, since $27 < 40$. We then follow the middle pointer, since $20 \leq 27 < 33$. For a range search, we find the first qualifying data entry as for an equality selection and then retrieve primary leaf pages sequentially (also retrieving overflow pages as needed by following pointers from the primary pages). The primary leaf pages are assumed to be allocated sequentially—this assumption is reasonable because the number of such pages is known when the tree is created and does not change subsequently under inserts and deletes—and so no ‘next leaf page’ pointers are needed.

We assume that each leaf page can contain two entries. If we now insert a record with key value 23, the entry $23^*$ belongs in the second data page, which already contains $20^*$ and $27^*$ and has no more space. We deal with this situation by adding an overflow page and putting $23^*$ in the overflow page. Chains of overflow pages can easily develop. For instance, inserting $48^*$, $41^*$, and $42^*$ leads to an overflow chain of two pages. The tree of Figure 9.5 with all these insertions is shown in Figure 9.6.

The deletion of an entry $k^*$ is handled by simply removing the entry. If this entry is on an overflow page and the overflow page becomes empty, the page can be removed. If the entry is on a primary page and deletion makes the primary page empty, the simplest approach is to simply leave the empty primary page as it is; it serves as a

<table>
<thead>
<tr>
<th>Data Pages</th>
<th>Index Pages</th>
<th>Overflow Pages</th>
</tr>
</thead>
</table>

**Figure 9.4** Page Allocation in ISAM
Figure 9.5 Sample ISAM Tree

Figure 9.6 ISAM Tree after Inserts
placeholder for future insertions (and possibly non-empty overflow pages, because we
do not move records from the overflow pages to the primary page when deletions on
the primary page create space). Thus, the number of primary leaf pages is fixed at file
creation time. Notice that deleting entries could lead to a situation in which key values
that appear in the index levels do not appear in the leaves! Since index levels are used
only to direct a search to the correct leaf page, this situation is not a problem. The
tree of Figure 9.6 is shown in Figure 9.7 after deletion of the entries 42*, 51*, and 97*.
Note that after deleting 51*, the key value 51 continues to appear in the index level.
A subsequent search for 51* would go to the correct leaf page and determine that the
entry is not in the tree.

The non-leaf pages direct a search to the correct leaf page. The number of disk I/Os
is equal to the number of levels of the tree and is equal to \( \log_F N \), where \( N \) is the
number of primary leaf pages and the fan-out \( F \) is the number of children per index
page. This number is considerably less than the number of disk I/Os for binary search,
which is \( \log_2 N \); in fact, it is reduced further by pinning the root page in memory.
The cost of access via a one-level index is \( \log_2(N/F) \). If we consider a file with 1,000,000
records, 10 records per leaf page, and 100 entries per index page, the cost (in page
I/Os) of a file scan is 100,000, a binary search of the sorted data file is 17, a binary
search of a one-level index is 10, and the ISAM file (assuming no overflow) is 3.

Note that once the ISAM file is created, inserts and deletes affect only the contents of
leaf pages. A consequence of this design is that long overflow chains could develop if a
number of inserts are made to the same leaf. These chains can significantly affect the
time to retrieve a record because the overflow chain has to be searched as well when
the search gets to this leaf. (Although data in the overflow chain can be kept sorted,
it usually is not, in order to make inserts fast.) To alleviate this problem, the tree
is initially created so that about 20 percent of each page is free. However, once the
free space is filled in with inserted records, unless space is freed again through deletes,
overflow chains can be eliminated only by a complete reorganization of the file.

The fact that only leaf pages are modified also has an important advantage with respect
to concurrent access. When a page is accessed, it is typically ‘locked’ by the requestor
to ensure that it is not concurrently modified by other users of the page. To modify
a page, it must be locked in ‘exclusive’ mode, which is permitted only when no one
else holds a lock on the page. Locking can lead to queues of users (transactions, to be
more precise) waiting to get access to a page. Queues can be a significant performance
bottleneck, especially for heavily accessed pages near the root of an index structure. In
the ISAM structure, since we know that index-level pages are never modified, we can
safely omit the locking step. Not locking index-level pages is an important advantage
of ISAM over a dynamic structure like a B+ tree. If the data distribution and size is
relatively static, which means overflow chains are rare, ISAM might be preferable to
B+ trees due to this advantage.

9.2 B+ TREES: A DYNAMIC INDEX STRUCTURE

A static structure such as the ISAM index suffers from the problem that long overflow
chains can develop as the file grows, leading to poor performance. This problem
motivated the development of more flexible, dynamic structures that adjust gracefully
to inserts and deletes. The B+ tree search structure, which is widely used, is a
balanced tree in which the internal nodes direct the search and the leaf nodes contain
the data entries. Since the tree structure grows and shrinks dynamically, it is not
feasible to allocate the leaf pages sequentially as in ISAM, where the set of primary
leaf pages was static. In order to retrieve all leaf pages efficiently, we have to link
them using page pointers. By organizing them into a doubly linked list, we can easily
traverse the sequence of leaf pages (sometimes called the sequence set) in either
direction. This structure is illustrated in Figure 9.8.

The following are some of the main characteristics of a B+ tree:

- Operations (insert, delete) on the tree keep it balanced.
- A minimum occupancy of 50 percent is guaranteed for each node except the root if
  the deletion algorithm discussed in Section 9.6 is implemented. However, deletion
  is often implemented by simply locating the data entry and removing it, without
  adjusting the tree as needed to guarantee the 50 percent occupancy, because files
typically grow rather than shrink.
- Searching for a record requires just a traversal from the root to the appropriate
  leaf. We will refer to the length of a path from the root to a leaf—any leaf, because
the tree is balanced—as the **height** of the tree. For example, a tree with only a leaf level and a single index level, such as the tree shown in Figure 9.10, has height 1. Because of high fan-out, the height of a B+ tree is rarely more than 3 or 4.

We will study B+ trees in which every node contains \( m \) entries, where \( d \leq m \leq 2d \). The value \( d \) is a parameter of the B+ tree, called the **order** of the tree, and is a measure of the capacity of a tree node. The root node is the only exception to this requirement on the number of entries; for the root it is simply required that \( 1 \leq m \leq 2d \).

If a file of records is updated frequently and sorted access is important, maintaining a B+ tree index with data records stored as data entries is almost always superior to maintaining a sorted file. For the space overhead of storing the index entries, we obtain all the advantages of a sorted file plus efficient insertion and deletion algorithms. B+ trees typically maintain 67 percent space occupancy. B+ trees are usually also preferable to ISAM indexing because inserts are handled gracefully without overflow chains. However, if the dataset size and distribution remain fairly static, overflow chains may not be a major problem. In this case, two factors favor ISAM: the leaf pages are allocated in sequence (making scans over a large range more efficient than in a B+ tree, in which pages are likely to get out of sequence on disk over time, even if they were in sequence after bulk-loading), and the locking overhead of ISAM is lower than that for B+ trees. As a general rule, however, B+ trees are likely to perform better than ISAM.

### 9.3 FORMAT OF A NODE

The format of a node is the same as for ISAM and is shown in Figure 9.1. Non-leaf nodes with \( m \) **index entries** contain \( m + 1 \) pointers to children. Pointer \( P_i \) points to a subtree in which all key values \( K \) are such that \( K_i \leq K < K_{i+1} \). As special cases, \( P_0 \) points to a tree in which all key values are less than \( K_1 \), and \( P_m \) points to a tree
in which all key values are greater than or equal to $K_m$. For leaf nodes, entries are denoted as $k^*$, as usual. Just as in ISAM, leaf nodes (and only leaf nodes!) contain data entries. In the common case that Alternative (2) or (3) is used, leaf entries are $\langle K, I(K) \rangle$ pairs, just like non-leaf entries. Regardless of the alternative chosen for leaf entries, the leaf pages are chained together in a doubly linked list. Thus, the leaves form a sequence, which can be used to answer range queries efficiently.

The reader should carefully consider how such a node organization can be achieved using the record formats presented in Section 7.7; after all, each key–pointer pair can be thought of as a record. If the field being indexed is of fixed length, these index entries will be of fixed length; otherwise, we have variable-length records. In either case the B+ tree can itself be viewed as a file of records. If the leaf pages do not contain the actual data records, then the B+ tree is indeed a file of records that is distinct from the file that contains the data. If the leaf pages contain data records, then a file contains the B+ tree as well as the data.

### 9.4 SEARCH

The algorithm for search finds the leaf node in which a given data entry belongs. A pseudocode sketch of the algorithm is given in Figure 9.9. We use the notation $*ptr$ to denote the value pointed to by a pointer variable $ptr$ and $& (value)$ to denote the address of $value$. Note that finding $i$ in tree_search requires us to search within the node, which can be done with either a linear search or a binary search (e.g., depending on the number of entries in the node).

In discussing the search, insertion, and deletion algorithms for B+ trees, we will assume that there are no duplicates. That is, no two data entries are allowed to have the same key value. Of course, duplicates arise whenever the search key does not contain a candidate key and must be dealt with in practice. We consider how duplicates can be handled in Section 9.7.

Consider the sample B+ tree shown in Figure 9.10. This B+ tree is of order $d=2$. That is, each node contains between 2 and 4 entries. Each non-leaf entry is a $\langle key\ value, nodepointer \rangle$ pair; at the leaf level, the entries are data records that we denote by $k^*$. To search for entry $5^*$, we follow the left-most child pointer, since $5 < 13$. To search for the entries $14^*$ or $15^*$, we follow the second pointer, since $13 \leq 14 < 17$, and $13 \leq 15 < 17$. (We don’t find $15^*$ on the appropriate leaf, and we can conclude that it is not present in the tree.) To find $24^*$, we follow the fourth child pointer, since $24 \leq 24 < 30$. 

func find (search key value \( K \)) returns nodepointer
// Given a search key value, finds its leaf node
return tree_search(root, \( K \)); // searches from root
endfunc

func tree_search (nodepointer, search key value \( K \)) returns nodepointer
// Searches tree for entry
if nodepointer is a leaf, return nodepointer;
else,
    if \( K < K_1 \) then return tree_search(\( P_0, K \));
    else,
        if \( K \geq K_m \) then return tree_search(\( P_m, K \)); // \( m = \# \) entries
        else,
            find \( i \) such that \( K_i \leq K < K_{i+1} \);
            return tree_search(\( P_i, K \))
        endfunc

Figure 9.9 Algorithm for Search

Root

Figure 9.10 Example of a B+ Tree, Order \( d=2 \)
9.5 INSERT

The algorithm for insertion takes an entry, finds the leaf node where it belongs, and inserts it there. Pseudocode for the B+ tree insertion algorithm is given in Figure 9.11. The basic idea behind the algorithm is that we recursively insert the entry by calling the insert algorithm on the appropriate child node. Usually, this procedure results in going down to the leaf node where the entry belongs, placing the entry there, and returning all the way back to the root node. Occasionally a node is full and it must be split. When the node is split, an entry pointing to the node created by the split must be inserted into its parent; this entry is pointed to by the pointer variable `newchildentry`. If the (old) root is split, a new root node is created and the height of the tree increases by one.

To illustrate insertion, let us continue with the sample tree shown in Figure 9.10. If we insert entry 8*, it belongs in the left-most leaf, which is already full. This insertion causes a split of the leaf page; the split pages are shown in Figure 9.12. The tree must now be adjusted to take the new leaf page into account, so we insert an entry consisting of the pair `(5, pointer to new page)` into the parent node. Notice how the key 5, which discriminates between the split leaf page and its newly created sibling, is ‘copied up.’ We cannot just ‘push up’ 5, because every data entry must appear in a leaf page.

Since the parent node is also full, another split occurs. In general we have to split a non-leaf node when it is full, containing $2d$ keys and $2d + 1$ pointers, and we have to add another index entry to account for a child split. We now have $2d + 1$ keys and $2d + 2$ pointers, yielding two minimally full non-leaf nodes, each containing $d$ keys and $d + 1$ pointers, and an extra key, which we choose to be the ‘middle’ key. This key and a pointer to the second non-leaf node constitute an index entry that must be inserted into the parent of the split non-leaf node. The middle key is thus ‘pushed up’ the tree, in contrast to the case for a split of a leaf page.

The split pages in our example are shown in Figure 9.13. The index entry pointing to the new non-leaf node is the pair `(17, pointer to new index-level page)`: notice that the key value 17 is ‘pushed up’ the tree, in contrast to the splitting key value 5 in the leaf split, which was ‘copied up.’

The difference in handling leaf-level and index-level splits arises from the B+ tree requirement that all data entries $k*$ must reside in the leaves. This requirement prevents us from ‘pushing up’ 5 and leads to the slight redundancy of having some key values appearing in the leaf level as well as in some index level. However, range queries can be efficiently answered by just retrieving the sequence of leaf pages; the redundancy is a small price to pay for efficiency. In dealing with the index levels, we have more flexibility, and we ‘push up’ 17 to avoid having two copies of 17 in the index levels.
proc insert (nodepointer, entry, newchildentry)
   // Inserts entry into subtree with root ‘*nodepointer’; degree is d;
   // ‘newchildentry’ is null initially, and null upon return unless child is split
   if *nodepointer is a non-leaf node, say N,
      find i such that $K_i \leq$ entry’s key value < $K_{i+1}$; // choose subtree
      insert($P_i$, entry, newchildentry); // recursively, insert entry
      if newchildentry is null, return; // usual case; didn’t split child
      else, // we split child, must insert *newchildentry in N
         if N has space, // usual case
            put *newchildentry on it, set newchildentry to null, return;
         else, // note difference wrt splitting of leaf page!
            split N: // 2$d+1$ key values and 2$d+2$ nodepointers
               first $d$ key values and $d+1$ nodepointers stay;
               last $d$ keys and $d+1$ pointers move to new node, $N2$;
               // *newchildentry set to guide searches between N and $N2$
               newchildentry = & (⟨smallest key value on $N2$, pointer to $N2$⟩);
               if N is the root, // root node was just split
                  create new node with (pointer to N, *newchildentry);
                  make the tree’s root-node pointer point to the new node;
               return;
   endproc

   if *nodepointer is a leaf node, say L,
      if L has space, // usual case
         put entry on it, set newchildentry to null, and return;
      else, // once in a while, the leaf is full
         split L: first $d$ entries stay, rest move to brand new node $L2$;
         newchildentry = & (⟨smallest key value on $L2$, pointer to $L2$⟩);
         set sibling pointers in L and $L2$;
         return;
endproc

Figure 9.11 Algorithm for Insertion into B+ Tree of Order $d$
Tree-Structured Indexing

Figure 9.12  Split Leaf Pages during Insert of Entry 8*

Entry to be inserted in parent node.
(Note that 5 is ‘copied up’ and continues to appear in the leaf.)

Figure 9.13  Split Index Pages during Insert of Entry 8*

Entry to be inserted in parent node.
(Note that 17 is ‘pushed up’ and appears once in the index. Contrast this with a leaf split.)

Now, since the split node was the old root, we need to create a new root node to hold the entry that distinguishes the two split index pages. The tree after completing the insertion of the entry 8* is shown in Figure 9.14.

Figure 9.14  B+ Tree after Inserting Entry 8*

One variation of the insert algorithm tries to redistribute entries of a node N with a sibling before splitting the node; this improves average occupancy. The sibling of a node N, in this context, is a node that is immediately to the left or right of N and has the same parent as N.

To illustrate redistribution, reconsider insertion of entry 8* into the tree shown in Figure 9.10. The entry belongs in the left-most leaf, which is full. However, the (only)
sibling of this leaf node contains only two entries and can thus accommodate more entries. We can therefore handle the insertion of 8* with a redistribution. Note how the entry in the parent node that points to the second leaf has a new key value; we ‘copy up’ the new low key value on the second leaf. This process is illustrated in Figure 9.15.

To determine whether redistribution is possible, we have to retrieve the sibling. If the sibling happens to be full, we have to split the node anyway. On average, checking whether redistribution is possible increases I/O for index node splits, especially if we check both siblings. (Checking whether redistribution is possible may reduce I/O if the redistribution succeeds whereas a split propagates up the tree, but this case is very infrequent.) If the file is growing, average occupancy will probably not be affected much even if we do not redistribute. Taking these considerations into account, not redistributing entries at non-leaf levels usually pays off.

If a split occurs at the leaf level, however, we have to retrieve a neighbor in order to adjust the previous and next-neighbor pointers with respect to the newly created leaf node. Therefore, a limited form of redistribution makes sense: If a leaf node is full, fetch a neighbor node; if it has space, and has the same parent, redistribute entries. Otherwise (neighbor has different parent, i.e., is not a sibling, or is also full) split the leaf node and adjust the previous and next-neighbor pointers in the split node, the newly created neighbor, and the old neighbor.

9.6 DELETE *

The algorithm for deletion takes an entry, finds the leaf node where it belongs, and deletes it. Pseudocode for the B+ tree deletion algorithm is given in Figure 9.16. The basic idea behind the algorithm is that we recursively delete the entry by calling the delete algorithm on the appropriate child node. We usually go down to the leaf node where the entry belongs, remove the entry from there, and return all the way back to the root node. Occasionally a node is at minimum occupancy before the deletion, and the deletion causes it to go below the occupancy threshold. When this happens,
we must either redistribute entries from an adjacent sibling or merge the node with a sibling to maintain minimum occupancy. If entries are redistributed between two nodes, their parent node must be updated to reflect this; the key value in the index entry pointing to the second node must be changed to be the lowest search key in the second node. If two nodes are merged, their parent must be updated to reflect this by deleting the index entry for the second node; this index entry is pointed to by the pointer variable \textit{oldchildentry} when the delete call returns to the parent node. If the last entry in the root node is deleted in this manner because one of its children was deleted, the height of the tree decreases by one.

To illustrate deletion, let us consider the sample tree shown in Figure 9.14. To delete entry 19*, we simply remove it from the leaf page on which it appears, and we are done because the leaf still contains two entries. If we subsequently delete 20*, however, the leaf contains only one entry after the deletion. The (only) sibling of the leaf node that contained 20* has three entries, and we can therefore deal with the situation by redistribution: we move entry 24* to the leaf page that contained 20* and ‘copy up’ the new splitting key (27, which is the new low key value of the leaf from which we borrowed 24*) into the parent. This process is illustrated in Figure 9.17.

Suppose that we now delete entry 24*. The affected leaf contains only one entry (22*) after the deletion, and the (only) sibling contains just two entries (27* and 29*). Therefore, we cannot redistribute entries. However, these two leaf nodes together contain only three entries and can be merged. While merging, we can ‘toss’ the entry \((27, \text{ pointer to second leaf page})\) in the parent, which pointed to the second leaf page, because the second leaf page is empty after the merge and can be discarded. The right subtree of Figure 9.17 after this step in the deletion of entry 24* is shown in Figure 9.18.

Deleting the entry \((27, \text{ pointer to second leaf page})\) has created a non-leaf-level page with just one entry, which is below the minimum of \(d=2\). To fix this problem, we must either redistribute or merge. In either case we must fetch a sibling. The only sibling of this node contains just two entries (with key values 5 and 13), and so redistribution is not possible; we must therefore merge.

The situation when we have to merge two non-leaf nodes is exactly the opposite of the situation when we have to split a non-leaf node. We have to split a non-leaf node when it contains \(2d\) keys and \(2d+1\) pointers, and we have to add another key–pointer pair. Since we resort to merging two non-leaf nodes only when we cannot redistribute entries between them, the two nodes must be minimally full; that is, each must contain \(d\) keys and \(d+1\) pointers prior to the deletion. After merging the two nodes and removing the key–pointer pair to be deleted, we have \(2d-1\) keys and \(2d+1\) pointers: Intuitively, the left-most pointer on the second merged node lacks a key value. To see what key value must be combined with this pointer to create a complete index entry, consider the parent of the two nodes being merged. The index entry pointing to one of the merged
**Figure 9.16** Algorithm for Deletion from B+ Tree of Order $d$
nodes must be deleted from the parent because the node is about to be discarded. The key value in this index entry is precisely the key value we need to complete the new merged node: The entries in the first node being merged, followed by the splitting key value that is ‘pulled down’ from the parent, followed by the entries in the second non-leaf node gives us a total of \(2d\) keys and \(2d + 1\) pointers, which is a full non-leaf node. Notice how the splitting key value in the parent is ‘pulled down,’ in contrast to the case of merging two leaf nodes.

Consider the merging of two non-leaf nodes in our example. Together, the non-leaf node and the sibling to be merged contain only three entries, and they have a total of five pointers to leaf nodes. To merge the two nodes, we also need to ‘pull down’ the index entry in their parent that currently discriminates between these nodes. This index entry has key value 17, and so we create a new entry \(\langle 17, \text{left-most child pointer in sibling} \rangle\). Now we have a total of four entries and five child pointers, which can fit on one page in a tree of order \(d=2\). Notice that pulling down the splitting key 17 means that it will no longer appear in the parent node following the merge. After we merge the affected non-leaf node and its sibling by putting all the entries on one page and discarding the empty sibling page, the new node is the only child of the old root, which can therefore be discarded. The tree after completing all these steps in the deletion of entry 24* is shown in Figure 9.19.
The previous examples illustrated redistribution of entries across leaves and merging of both leaf-level and non-leaf-level pages. The remaining case is that of redistribution of entries between non-leaf-level pages. To understand this case, consider the intermediate right subtree shown in Figure 9.18. We would arrive at the same intermediate right subtree if we try to delete 24* from a tree similar to the one shown in Figure 9.17 but with the left subtree and root key value as shown in Figure 9.20. The tree in Figure 9.20 illustrates an intermediate stage during the deletion of 24*. (Try to construct the initial tree.)

In contrast to the case when we deleted 24* from the tree of Figure 9.17, the non-leaf level node containing key value 30 now has a sibling that can spare entries (the entries with key values 17 and 20). We move these entries\(^2\) over from the sibling. Notice that in doing so, we essentially ‘push’ them through the splitting entry in their parent node (the root), which takes care of the fact that 17 becomes the new low key value on the right and therefore must replace the old splitting key in the root (the key value 22). The tree with all these changes is shown in Figure 9.21.

In concluding our discussion of deletion, we note that we retrieve only one sibling of a node. If this node has spare entries, we use redistribution; otherwise, we merge. If the node has a second sibling, it may be worth retrieving that sibling as well to

\(^2\)It is sufficient to move over just the entry with key value 20, but we are moving over two entries to illustrate what happens when several entries are redistributed.
check for the possibility of redistribution. Chances are high that redistribution will be possible, and unlike merging, redistribution is guaranteed to propagate no further than the parent node. Also, the pages have more space on them, which reduces the likelihood of a split on subsequent insertions. (Remember, files typically grow, not shrink!) However, the number of times that this case arises (node becomes less than half-full and first sibling can’t spare an entry) is not very high, so it is not essential to implement this refinement of the basic algorithm that we have presented.

9.7 DUPLICATES *

The search, insertion, and deletion algorithms that we have presented ignore the issue of duplicate keys, that is, several data entries with the same key value. We now discuss how duplicates can be handled.

The basic search algorithm assumes that all entries with a given key value reside on a single leaf page. One way to satisfy this assumption is to use overflow pages to deal with duplicates. (In ISAM, of course, we have overflow pages in any case, and duplicates are easily handled.)

Typically, however, we use an alternative approach for duplicates. We handle them just like any other entries and several leaf pages may contain entries with a given key value. To retrieve all data entries with a given key value, we must search for the left-most data entry with the given key value and then possibly retrieve more than one leaf page (using the leaf sequence pointers). Modifying the search algorithm to find the left-most data entry in an index with duplicates is an interesting exercise (in fact, it is Exercise 9.11).

One problem with this approach is that when a record is deleted, if we use Alternative (2) for data entries, finding the corresponding data entry to delete in the B+ tree index could be inefficient because we may have to check several duplicate entries \( \langle key, rid \rangle \) with the same key value. This problem can be addressed by considering the \( rid \) value in the data entry to be part of the search key, for purposes of positioning the data.
Duplicate handling in commercial systems: In a clustered index in Sybase ASE, the data rows are maintained in sorted order on the page and in the collection of data pages. The data pages are bidirectionally linked in sort order. Rows with duplicate keys are inserted into (or deleted from) the ordered set of rows. This may result in overflow pages of rows with duplicate keys being inserted into the page chain or empty overflow pages removed from the page chain. Insertion or deletion of a duplicate key does not affect the higher index levels unless a split or merge of a non-overflow page occurs. In IBM DB2, Oracle 8, and Microsoft SQL Server, duplicates are handled by adding a row id if necessary to eliminate duplicate key values.

entry in the tree. This solution effectively turns the index into a unique index (i.e., no duplicates). Remember that a search key can be any sequence of fields—in this variant, the rid of the data record is essentially treated as another field while constructing the search key.

Alternative (3) for data entries leads to a natural solution for duplicates, but if we have a large number of duplicates, a single data entry could span multiple pages. And of course, when a data record is deleted, finding the rid to delete from the corresponding data entry can be inefficient. The solution to this problem is similar to the one discussed above for Alternative (2): We can maintain the list of rids within each data entry in sorted order (say, by page number and then slot number if a rid consists of a page id and a slot id).

9.8 B+ TREES IN PRACTICE *

In this section we discuss several important pragmatic issues.

9.8.1 Key Compression

The height of a B+ tree depends on the number of data entries and the size of index entries. The size of index entries determines the number of index entries that will fit on a page and, therefore, the fan-out of the tree. Since the height of the tree is proportional to \( \log_{\text{fan-out}}(\# \text{ of data entries}) \), and the number of disk I/Os to retrieve a data entry is equal to the height (unless some pages are found in the buffer pool) it is clearly important to maximize the fan-out, to minimize the height.

An index entry contains a search key value and a page pointer. Thus the size primarily depends on the size of the search key value. If search key values are very long (for instance, the name Devarakonda Venkataramana Sathyanarayana Seshasayee Yella-
B+ Trees in Real Systems: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all support clustered and unclustered B+ tree indexes, with some differences in how they handle deletions and duplicate key values. In Sybase ASE, depending on the concurrency control scheme being used for the index, the deleted row is removed (with merging if the page occupancy goes below threshold) or simply marked as deleted; a garbage collection scheme is used to recover space in the latter case. In Oracle 8, deletions are handled by marking the row as deleted. To reclaim the space occupied by deleted records, we can rebuild the index online (i.e., while users continue to use the index) or coalesce underfull pages (which does not reduce tree height). Coalesce is in-place, rebuild creates a copy. Informix handles deletions by marking simply marking records as deleted. DB2 and SQL Server remove deleted records and merge pages when occupancy goes below threshold.

Oracle 8 also allows records from multiple relations to be co-clustered on the same page. The co-clustering can be based on a B+ tree search key or static hashing and upto 32 relns can be stored together.

manchali Murthy), not many index entries will fit on a page; fan-out is low, and the height of the tree is large.

On the other hand, search key values in index entries are used only to direct traffic to the appropriate leaf. When we want to locate data entries with a given search key value, we compare this search key value with the search key values of index entries (on a path from the root to the desired leaf). During the comparison at an index-level node, we want to identify two index entries with search key values $k_1$ and $k_2$ such that the desired search key value $k$ falls between $k_1$ and $k_2$. To accomplish this, we do not need to store search key values in their entirety in index entries.

For example, suppose that we have two adjacent index entries in a node, with search key values ‘David Smith’ and ‘Devarakonda . . . ’. To discriminate between these two values, it is sufficient to store the abbreviated forms ‘Da’ and ‘De.’ More generally, the meaning of the entry ‘David Smith’ in the B+ tree is that every value in the subtree pointed to by the pointer to the left of ‘David Smith’ is less than ‘David Smith,’ and every value in the subtree pointed to by the pointer to the right of ‘David Smith’ is (greater than or equal to ‘David Smith’ and) less than ‘Devarakonda . . . ’.

To ensure that this semantics for an entry is preserved, while compressing the entry with key ‘David Smith,’ we must examine the largest key value in the subtree to the left of ‘David Smith’ and the smallest key value in the subtree to the right of ‘David Smith,’ not just the index entries (‘Daniel Lee’ and ‘Devarakonda . . . ’) that are its neighbors. This point is illustrated in Figure 9.22; the value ‘Davey Jones’ is greater than ‘Dav,’ and thus, ‘David Smith’ can only be abbreviated to ‘Davi,’ not to ‘Dav.’
This technique is called prefix key compression, or simply key compression, and is supported in many commercial implementations of B+ trees. It can substantially increase the fan-out of a tree. We will not discuss the details of the insertion and deletion algorithms in the presence of key compression.

### 9.8.2 Bulk-Loading a B+ Tree

Entries are added to a B+ tree in two ways. First, we may have an existing collection of data records with a B+ tree index on it; whenever a record is added to the collection, a corresponding entry must be added to the B+ tree as well. (Of course, a similar comment applies to deletions.) Second, we may have a collection of data records for which we want to create a B+ tree index on some key field(s). In this situation, we can start with an empty tree and insert an entry for each data record, one at a time, using the standard insertion algorithm. However, this approach is likely to be quite expensive because each entry requires us to start from the root and go down to the appropriate leaf page. Even though the index-level pages are likely to stay in the buffer pool between successive requests, the overhead is still considerable.

For this reason many systems provide a bulk-loading utility for creating a B+ tree index on an existing collection of data records. The first step is to sort the data entries $k*$ to be inserted into the (to be created) B+ tree according to the search key $k$. (If the entries are key-pointer pairs, sorting them does not mean sorting the data records that are pointed to, of course.) We will use a running example to illustrate the bulk-loading algorithm. We will assume that each data page can hold only two entries, and that each index page can hold two entries and an additional pointer (i.e., the B+ tree is assumed to be of order $d=1$).

After the data entries have been sorted, we allocate an empty page to serve as the root and insert a pointer to the first page of (sorted) entries into it. We illustrate this process in Figure 9.23, using a sample set of nine sorted pages of data entries.
We then add one entry to the root page for each page of the sorted data entries. The new entry consists of \(\langle\text{low key value on page, pointer to page}\rangle\). We proceed until the root page is full; see Figure 9.24.

To insert the entry for the next page of data entries, we must split the root and create a new root page. We show this step in Figure 9.25.
We have redistributed the entries evenly between the two children of the root, in anticipation of the fact that the B+ tree is likely to grow. Although it is difficult (!) to illustrate these options when at most two entries fit on a page, we could also have just left all the entries on the old page or filled up some desired fraction of that page (say, 80 percent). These alternatives are simple variants of the basic idea.

To continue with the bulk-loading example, entries for the leaf pages are always inserted into the right-most index page just above the leaf level. When the right-most index page above the leaf level fills up, it is split. This action may cause a split of the right-most index page one step closer to the root, as illustrated in Figures 9.26 and 9.27.

**Figure 9.26**  Before Adding Entry for Leaf Page Containing 38*

**Figure 9.27**  After Adding Entry for Leaf Page Containing 38*
Note that splits occur only on the right-most path from the root to the leaf level. We leave the completion of the bulk-loading example as a simple exercise.

Let us consider the cost of creating an index on an existing collection of records. This operation consists of three steps: (1) creating the data entries to insert in the index, (2) sorting the data entries, and (3) building the index from the sorted entries. The first step involves scanning the records and writing out the corresponding data entries; the cost is \((R + E)\) I/Os, where \(R\) is the number of pages containing records and \(E\) is the number of pages containing data entries. Sorting is discussed in Chapter 11; you will see that the index entries can be generated in sorted order at a cost of about \(3E\) I/Os. These entries can then be inserted into the index as they are generated, using the bulk-loading algorithm discussed in this section. The cost of the third step, that is, inserting the entries into the index, is then just the cost of writing out all index pages.

### 9.8.3 The Order Concept

We have presented B+ trees using the parameter \(d\) to denote minimum occupancy. It is worth noting that the concept of order (i.e., the parameter \(d\)), while useful for teaching B+ tree concepts, must usually be relaxed in practice and replaced by a physical space criterion; for example, that nodes must be kept at least half-full.

One reason for this is that leaf nodes and non-leaf nodes can usually hold different numbers of entries. Recall that B+ tree nodes are disk pages and that non-leaf nodes contain only search keys and node pointers, while leaf nodes can contain the actual data records. Obviously, the size of a data record is likely to be quite a bit larger than the size of a search entry, so many more search entries than records will fit on a disk page.

A second reason for relaxing the order concept is that the search key may contain a character string field (e.g., the name field of Students) whose size varies from record to record; such a search key leads to variable-size data entries and index entries, and the number of entries that will fit on a disk page becomes variable.

Finally, even if the index is built on a fixed-size field, several records may still have the same search key value (e.g., several Students records may have the same gpa or name value). This situation can also lead to variable-size leaf entries (if we use Alternative (3) for data entries). Because of all of these complications, the concept of order is typically replaced by a simple physical criterion (e.g., merge if possible when more than half of the space in the node is unused).
9.8.4 The Effect of Inserts and Deletes on Rids

If the leaf pages contain data records—that is, the B+ tree is a clustered index—then operations such as splits, merges, and redistributions can change rids. Recall that a typical representation for a rid is some combination of (physical) page number and slot number. This scheme allows us to move records within a page if an appropriate page format is chosen, but not across pages, as is the case with operations such as splits. So unless rids are chosen to be independent of page numbers, an operation such as split or merge in a clustered B+ tree may require compensating updates to other indexes on the same data.

A similar comment holds for any dynamic clustered index, regardless of whether it is tree-based or hash-based. Of course, the problem does not arise with nonclustered indexes because only index entries are moved around.

9.9 POINTS TO REVIEW

- Tree-structured indexes are ideal for range selections, and also support equality selections quite efficiently. *ISAM* is a static tree-structured index in which only leaf pages are modified by inserts and deletes. If a leaf page is full, an *overflow page* is added. Unless the size of the dataset and the data distribution remain approximately the same, overflow chains could become long and degrade performance. *(Section 9.1)*

- A B+ tree is a dynamic, height-balanced index structure that adapts gracefully to changing data characteristics. Each node except the root has between $d$ and $2d$ entries. The number $d$ is called the *order* of the tree. *(Section 9.2)*

- Each non-leaf node with $m$ index entries has $m+1$ children pointers. The leaf nodes contain data entries. Leaf pages are chained in a doubly linked list. *(Section 9.3)*

- An equality search requires traversal from the root to the corresponding leaf node of the tree. *(Section 9.4)*

- During insertion, nodes that are full are *split* to avoid overflow pages. Thus, an insertion might increase the height of the tree. *(Section 9.5)*

- During deletion, a node might go below the minimum occupancy threshold. In this case, we can either redistribute entries from adjacent siblings, or we can merge the node with a sibling node. A deletion might decrease the height of the tree. *(Section 9.6)*

- Duplicate search keys require slight modifications to the basic B+ tree operations. *(Section 9.7)*
In key compression, search key values in index nodes are shortened to ensure a high fan-out. A new B+ tree index can be efficiently constructed for a set of records using a bulk-loading procedure. In practice, the concept of order is replaced by a physical space criterion. *(Section 9.8)*

**EXERCISES**

**Exercise 9.1** Consider the B+ tree index of order \( d = 2 \) shown in Figure 9.28.

1. Show the tree that would result from inserting a data entry with key 9 into this tree.
2. Show the B+ tree that would result from inserting a data entry with key 3 into the original tree. How many page reads and page writes will the insertion require?
3. Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the left sibling is checked for possible redistribution.
4. Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the right sibling is checked for possible redistribution.
5. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 46 and then deleting the data entry with key 52.
6. Show the B+ tree that would result from deleting the data entry with key 91 from the original tree.
7. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 59, and then deleting the data entry with key 91.
8. Show the B+ tree that would result from successively deleting the data entries with keys 32, 39, 41, 45, and 73 from the original tree.

**Exercise 9.2** Consider the B+ tree index shown in Figure 9.29, which uses Alternative (1) for data entries. Each intermediate node can hold up to five pointers and four key values. Each leaf can hold up to four records, and leaf nodes are doubly linked as usual, although these links are not shown in the figure.

Answer the following questions.

1. Name all the tree nodes that must be fetched to answer the following query: “Get all records with search key greater than 38.”
2. Insert a record with search key 109 into the tree.
3. Delete the record with search key 81 from the (original) tree.
4. Name a search key value such that inserting it into the (original) tree would cause an increase in the height of the tree.
5. Note that subtrees A, B, and C are not fully specified. Nonetheless, what can you infer about the contents and the shape of these trees?
6. How would your answers to the above questions change if this were an ISAM index?
7. Suppose that this is an ISAM index. What is the minimum number of insertions needed to create a chain of three overflow pages?

**Exercise 9.3** Answer the following questions.

1. What is the minimum space utilization for a B+ tree index?
2. What is the minimum space utilization for an ISAM index?
3. If your database system supported both a static and a dynamic tree index (say, ISAM and B+ trees), would you ever consider using the static index in preference to the dynamic index?

**Exercise 9.4** Suppose that a page can contain at most four data values and that all data values are integers. Using only B+ trees of order 2, give examples of each of the following:

1. A B+ tree whose height changes from 2 to 3 when the value 25 is inserted. Show your structure before and after the insertion.
2. A B+ tree in which the deletion of the value 25 leads to a redistribution. Show your structure before and after the deletion.
3. A B+ tree in which the deletion of the value 25 causes a merge of two nodes, but without altering the height of the tree.

4. An ISAM structure with four buckets, none of which has an overflow page. Further, every bucket has space for exactly one more entry. Show your structure before and after inserting two additional values, chosen so that an overflow page is created.

**Exercise 9.5** Consider the B+ tree shown in Figure 9.30.

1. Identify a list of five data entries such that:
   
   (a) Inserting the entries in the order shown and then deleting them in the opposite order (e.g., insert $a$, insert $b$, delete $b$, delete $a$) results in the original tree.
   
   (b) Inserting the entries in the order shown and then deleting them in the opposite order (e.g., insert $a$, insert $b$, delete $b$, delete $a$) results in a different tree.

2. What is the minimum number of insertions of data entries with distinct keys that will cause the height of the (original) tree to change from its current value (of 1) to 3?

3. Would the minimum number of insertions that will cause the original tree to increase to height 3 change if you were allowed to insert duplicates (multiple data entries with the same key), assuming that overflow pages are not used for handling duplicates?

**Exercise 9.6** Answer Exercise 9.5 assuming that the tree is an ISAM tree! (Some of the examples asked for may not exist—if so, explain briefly.)

**Exercise 9.7** Suppose that you have a sorted file, and you want to construct a dense primary B+ tree index on this file.

1. One way to accomplish this task is to scan the file, record by record, inserting each one using the B+ tree insertion procedure. What performance and storage utilization problems are there with this approach?

2. Explain how the bulk-loading algorithm described in the text improves upon the above scheme.

**Exercise 9.8** Assume that you have just built a dense B+ tree index using Alternative (2) on a heap file containing 20,000 records. The key field for this B+ tree index is a 40-byte string, and it is a candidate key. Pointers (i.e., record ids and page ids) are (at most) 10-byte values. The size of one disk page is 1,000 bytes. The index was built in a bottom-up fashion using the bulk-loading algorithm, and the nodes at each level were filled up as much as possible.
1. How many levels does the resulting tree have?
2. For each level of the tree, how many nodes are at that level?
3. How many levels would the resulting tree have if key compression is used and it reduces the average size of each key in an entry to 10 bytes?
4. How many levels would the resulting tree have without key compression, but with all pages 70 percent full?

**Exercise 9.9** The algorithms for insertion and deletion into a B+ tree are presented as recursive algorithms. In the code for `insert`, for instance, there is a call made at the parent of a node N to insert into (the subtree rooted at) node N, and when this call returns, the current node is the parent of N. Thus, we do not maintain any ‘parent pointers’ in nodes of B+ tree. Such pointers are not part of the B+ tree structure for a good reason, as this exercise will demonstrate. An alternative approach that uses parent pointers—again, remember that such pointers are *not* part of the standard B+ tree structure!—in each node appears to be simpler:

Search to the appropriate leaf using the search algorithm; then insert the entry and split if necessary, with splits propagated to parents if necessary (using the parent pointers to find the parents).

Consider this (unsatisfactory) alternative approach:

1. Suppose that an internal node N is split into nodes N and N2. What can you say about the parent pointers in the children of the original node N?
2. Suggest two ways of dealing with the inconsistent parent pointers in the children of node N.
3. For each of the above suggestions, identify a potential (major) disadvantage.
4. What conclusions can you draw from this exercise?

**Exercise 9.10** Consider the instance of the Students relation shown in Figure 9.31. Show a B+ tree of order 2 in each of these cases, assuming that duplicates are handled using overflow pages. Clearly indicate what the data entries are (i.e., do not use the ‘k∗’ convention).

1. A dense B+ tree index on `age` using Alternative (1) for data entries.
2. A sparse B+ tree index on `age` using Alternative (1) for data entries.
3. A dense B+ tree index on `gpa` using Alternative (2) for data entries. For the purposes of this question, assume that these tuples are stored in a sorted file in the order shown in the figure: the first tuple is in page 1, slot 1; the second tuple is in page 1, slot 2; and so on. Each page can store up to three data records. You can use `<page-id, slot>` to identify a tuple.

**Exercise 9.11** Suppose that duplicates are handled using the approach without overflow pages discussed in Section 9.7. Describe an algorithm to search for the left-most occurrence of a data entry with search key value K.

**Exercise 9.12** Answer Exercise 9.10 assuming that duplicates are handled without using overflow pages, using the alternative approach suggested in Section 9.7.
Tree-Structured Indexing

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Madayan</td>
<td>madayan@music</td>
<td>11</td>
<td>1.8</td>
</tr>
<tr>
<td>53832</td>
<td>Guldu</td>
<td>guldu@music</td>
<td>12</td>
<td>3.8</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53901</td>
<td>Jones</td>
<td>jones@toy</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53902</td>
<td>Jones</td>
<td>jones@physics</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53903</td>
<td>Jones</td>
<td>jones@english</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53904</td>
<td>Jones</td>
<td>jones@genetics</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53905</td>
<td>Jones</td>
<td>jones@astro</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53906</td>
<td>Jones</td>
<td>jones@chem</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53902</td>
<td>Jones</td>
<td>jones@sanitation</td>
<td>18</td>
<td>3.8</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>19</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
<tr>
<td>54001</td>
<td>Smith</td>
<td>smith@ee</td>
<td>19</td>
<td>3.5</td>
</tr>
<tr>
<td>54005</td>
<td>Smith</td>
<td>smith@cs</td>
<td>19</td>
<td>3.8</td>
</tr>
<tr>
<td>54009</td>
<td>Smith</td>
<td>smith@astro</td>
<td>19</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Figure 9.31  An Instance of the Students Relation

PROJECT-BASED EXERCISES

Exercise 9.13 Compare the public interfaces for heap files, B+ tree indexes, and linear hashed indexes. What are the similarities and differences? Explain why these similarities and differences exist.

Exercise 9.14 This exercise involves using Minibase to explore the earlier (non-project) exercises further.

1. Create the trees shown in earlier exercises and visualize them using the B+ tree visualizer in Minibase.

2. Verify your answers to exercises that require insertion and deletion of data entries by doing the insertions and deletions in Minibase and looking at the resulting trees using the visualizer.

Exercise 9.15 (Note to instructors: Additional details must be provided if this exercise is assigned; see Appendix B.) Implement B+ trees on top of the lower-level code in Minibase.

BIBLIOGRAPHIC NOTES

The original version of the B+ tree was presented by Bayer and McCreight [56]. The B+ tree is described in [381] and [163]. B tree indexes for skewed data distributions are studied in [222]. The VSAM indexing structure is described in [671]. Various tree structures for supporting range queries are surveyed in [66]. An early paper on multiattribute search keys is [433].

References for concurrent access to B trees are in the bibliography for Chapter 19.
In this chapter we consider file organizations that are excellent for equality selections. The basic idea is to use a hashing function, which maps values in a search field into a range of bucket numbers to find the page on which a desired data entry belongs. We use a simple scheme called Static Hashing to introduce the idea. This scheme, like ISAM, suffers from the problem of long overflow chains, which can affect performance. Two solutions to the problem are presented. The Extendible Hashing scheme uses a directory to support inserts and deletes efficiently without any overflow pages. The Linear Hashing scheme uses a clever policy for creating new buckets and supports inserts and deletes efficiently without the use of a directory. Although overflow pages are used, the length of overflow chains is rarely more than two.

Hash-based indexing techniques cannot support range searches, unfortunately. Tree-based indexing techniques, discussed in Chapter 9, can support range searches efficiently and are almost as good as hash-based indexing for equality selections. Thus, many commercial systems choose to support only tree-based indexes. Nonetheless, hashing techniques prove to be very useful in implementing relational operations such as joins, as we will see in Chapter 12. In particular, the Index Nested Loops join method generates many equality selection queries, and the difference in cost between a hash-based index and a tree-based index can become significant in this context.

The rest of this chapter is organized as follows. Section 10.1 presents Static Hashing. Like ISAM, its drawback is that performance degrades as the data grows and shrinks. We discuss a dynamic hashing technique called Extendible Hashing in Section 10.2 and another dynamic technique, called Linear Hashing, in Section 10.3. We compare Extendible and Linear Hashing in Section 10.4.

10.1 STATIC HASHING

The Static Hashing scheme is illustrated in Figure 10.1. The pages containing the data can be viewed as a collection of buckets, with one primary page and possibly
additional **overflow** pages per bucket. A file consists of buckets 0 through \( N - 1 \), with one primary page per bucket initially. Buckets contain **data entries**, which can be any of the three alternatives discussed in Chapter 8.

![Diagram of Static Hashing](image)

**Figure 10.1** Static Hashing

To search for a data entry, we apply a **hash function** \( h \) to identify the bucket to which it belongs and then search this bucket. To speed the search of a bucket, we can maintain data entries in sorted order by search key value; in this chapter, we do not sort entries, and the order of entries within a bucket has no significance. In order to insert a data entry, we use the hash function to identify the correct bucket and then put the data entry there. If there is no space for this data entry, we allocate a new **overflow** page, put the data entry on this page, and add the page to the **overflow chain** of the bucket. To delete a data entry, we use the hashing function to identify the correct bucket, locate the data entry by searching the bucket, and then remove it. If this data entry is the last in an overflow page, the overflow page is removed from the overflow chain of the bucket and added to a list of **free pages**.

The hash function is an important component of the hashing approach. It must distribute values in the domain of the search field uniformly over the collection of buckets. If we have \( N \) buckets, numbered 0 through \( N - 1 \), a hash function \( h \) of the form \( h(value) = (a \times value + b) \) works well in practice. (The bucket identified is \( h(value) \mod N \).) The constants \( a \) and \( b \) can be chosen to ‘tune’ the hash function.

Since the number of buckets in a Static Hashing file is known when the file is created, the primary pages can be stored on successive disk pages. Thus, a search ideally requires just one disk I/O, and insert and delete operations require two I/Os (read and write the page), although the cost could be higher in the presence of overflow pages. As the file grows, long overflow chains can develop. Since searching a bucket requires us to search (in general) all pages in its overflow chain, it is easy to see how performance can deteriorate. By initially keeping pages 80 percent full, we can avoid overflow pages if the file doesn’t grow too much, but in general the only way to get rid of overflow chains is to create a new file with more buckets.
The main problem with Static Hashing is that the number of buckets is fixed. If a file shrinks greatly, a lot of space is wasted; more importantly, if a file grows a lot, long overflow chains develop, resulting in poor performance. One alternative is to periodically ‘rehash’ the file to restore the ideal situation (no overflow chains, about 80 percent occupancy). However, rehashing takes time and the index cannot be used while rehashing is in progress. Another alternative is to use dynamic hashing techniques such as Extendible and Linear Hashing, which deal with inserts and deletes gracefully. We consider these techniques in the rest of this chapter.

10.1.1 Notation and Conventions

In the rest of this chapter, we use the following conventions. The first step in searching for, inserting, or deleting a data entry \( k^* \) (with search key \( k \)) is always to apply a hash function \( h \) to the search field, and we will denote this operation as \( h(k) \). The value \( h(k) \) identifies a bucket. We will often denote the data entry \( k^* \) by using the hash value, as \( h(k)^* \). Note that two different keys can have the same hash value.

10.2 EXTENDIBLE HASHING *

To understand Extendible Hashing, let us begin by considering a Static Hashing file. If we have to insert a new data entry into a full bucket, we need to add an overflow page. If we don’t want to add overflow pages, one solution is to reorganize the file at this point by doubling the number of buckets and redistributing the entries across the new set of buckets. This solution suffers from one major defect—the entire file has to be read, and twice as many pages have to be written, to achieve the reorganization. This problem, however, can be overcome by a simple idea: use a directory of pointers to buckets, and double the size of the number of buckets by doubling just the directory and splitting only the bucket that overflowed.

To understand the idea, consider the sample file shown in Figure 10.2. The directory consists of an array of size 4, with each element being a pointer to a bucket. (The global depth and local depth fields will be discussed shortly; ignore them for now.) To locate a data entry, we apply a hash function to the search field and take the last two bits of its binary representation to get a number between 0 and 3. The pointer in this array position gives us the desired bucket; we assume that each bucket can hold four data entries. Thus, to locate a data entry with hash value 5 (binary 101), we look at directory element 01 and follow the pointer to the data page (bucket B in the figure).

To insert a data entry, we search to find the appropriate bucket. For example, to insert a data entry with hash value 13 (denoted as 13*), we would examine directory element 01 and go to the page containing data entries 1*, 5*, and 21*. Since the page has space for an additional data entry, we are done after we insert the entry (Figure 10.3).
Hash-Based Indexing

Figure 10.2  Example of an Extendible Hashed File

Figure 10.3  After Inserting Entry r with h(r)=13
Next, let us consider insertion of a data entry into a full bucket. The essence of the Extendible Hashing idea lies in how we deal with this case. Consider the insertion of data entry 20* (binary 10100). Looking at directory element 00, we are led to bucket A, which is already full. We must first split the bucket by allocating a new bucket and redistributing the contents (including the new entry to be inserted) across the old bucket and its 'split image.' To redistribute entries across the old bucket and its split image, we consider the last three bits of \( h(r) \); the last two bits are 00, indicating a data entry that belongs to one of these two buckets, and the third bit discriminates between these buckets. The redistribution of entries is illustrated in Figure 10.4.

![Figure 10.4](image-url)

Figure 10.4 While Inserting Entry \( r \) with \( h(r)=20 \)

Notice a problem that we must now resolve—we need three bits to discriminate between two of our data pages (A and A2), but the directory has only enough slots to store all two-bit patterns. The solution is to double the directory. Elements that differ only in the third bit from the end are said to ‘correspond’: corresponding elements of the directory point to the same bucket with the exception of the elements corresponding to the split bucket. In our example, bucket 0 was split; so, new directory element 000 points to one of the split versions and new element 100 points to the other. The sample file after completing all steps in the insertion of 20* is shown in Figure 10.5.

Thus, doubling the file requires allocating a new bucket page, writing both this page and the old bucket page that is being split, and doubling the directory array.

---

1 Since there are no overflow pages in Extendible Hashing, a bucket can be thought of as a single page.
directory is likely to be much smaller than the file itself because each element is just a page-id, and can be doubled by simply copying it over (and adjusting the elements for the split buckets). The cost of doubling is now quite acceptable.

We observe that the basic technique used in Extendible Hashing is to treat the result of applying a hash function \( h \) as a binary number and to interpret the last \( d \) bits, where \( d \) depends on the size of the directory, as an offset into the directory. In our example \( d \) is originally 2 because we only have four buckets; after the split, \( d \) becomes 3 because we now have eight buckets. A corollary is that when distributing entries across a bucket and its split image, we should do so on the basis of the \( d \)th bit. (Note how entries are redistributed in our example; see Figure 10.5.) The number \( d \) is called the **global depth** of the hashed file and is kept as part of the header of the file. It is used every time we need to locate a data entry.

An important point that arises is whether splitting a bucket necessitates a directory doubling. Consider our example, as shown in Figure 10.5. If we now insert 9*, it belongs in bucket B; this bucket is already full. We can deal with this situation by splitting the bucket and using directory elements 001 and 101 to point to the bucket and its split image, as shown in Figure 10.6.

Thus, a bucket split does not necessarily require a directory doubling. However, if either bucket A or A2 grows full and an insert then forces a bucket split, we are forced to double the directory again.
In order to differentiate between these cases, and determine whether a directory doubling is needed, we maintain a local depth for each bucket. If a bucket whose local depth is equal to the global depth is split, the directory must be doubled. Going back to the example, when we inserted 9* into the index shown in Figure 10.5, it belonged to bucket B with local depth 2, whereas the global depth was 3. Even though the bucket was split, the directory did not have to be doubled. Buckets A and A2, on the other hand, have local depth equal to the global depth and, if they grow full and are split, the directory must then be doubled.

Initially, all local depths are equal to the global depth (which is the number of bits needed to express the total number of buckets). We increment the global depth by 1 each time the directory doubles, of course. Also, whenever a bucket is split (whether or not the split leads to a directory doubling), we increment by 1 the local depth of the split bucket and assign this same (incremented) local depth to its (newly created) split image. Intuitively, if a bucket has local depth $l$, the hash values of data entries in it agree upon the last $l$ bits; further, no data entry in any other bucket of the file has a hash value with the same last $l$ bits. A total of $2^{d-l}$ directory elements point to a bucket with local depth $l$; if $d = l$, exactly one directory element is pointing to the bucket, and splitting such a bucket requires directory doubling.
A final point to note is that we can also use the first $d$ bits (the most significant bits) instead of the last $d$ (least significant bits), but in practice the last $d$ bits are used. The reason is that a directory can then be doubled simply by copying it.

In summary, a data entry can be located by computing its hash value, taking the last $d$ bits, and looking in the bucket pointed to by this directory element. For inserts, the data entry is placed in the bucket to which it belongs and the bucket is split if necessary to make space. A bucket split leads to an increase in the local depth, and if the local depth becomes greater than the global depth as a result, to a directory doubling (and an increase in the global depth) as well.

For deletes, the data entry is located and removed. If the delete leaves the bucket empty, it can be merged with its split image, although this step is often omitted in practice. Merging buckets decreases the local depth. If each directory element points to the same bucket as its split image (i.e., $0$ and $2^d - 1$ point to the same bucket, namely A; $1$ and $2^d - 1 + 1$ point to the same bucket, namely B, which may or may not be identical to A; etc.), we can halve the directory and reduce the global depth, although this step is not necessary for correctness.

The insertion examples can be worked out backwards as examples of deletion. (Start with the structure shown after an insertion and delete the inserted element. In each case the original structure should be the result.)

If the directory fits in memory, an equality selection can be answered in a single disk access, as for Static Hashing (in the absence of overflow pages), but otherwise, two disk I/Os are needed. As a typical example, a 100 MB file with 100 bytes per data entry and a page size of 4 KB contains 1,000,000 data entries and only about 25,000 elements in the directory. (Each page/bucket contains roughly 40 data entries, and we have one directory element per bucket.) Thus, although equality selections can be twice as slow as for Static Hashing files, chances are high that the directory will fit in memory and performance is the same as for Static Hashing files.

On the other hand, the directory grows in spurts and can become large for skewed data distributions (where our assumption that data pages contain roughly equal numbers of data entries is not valid). In the context of hashed files, a skewed data distribution is one in which the distribution of hash values of search field values (rather than the distribution of search field values themselves) is skewed (very ‘bursty’ or nonuniform). Even if the distribution of search values is skewed, the choice of a good hashing function typically yields a fairly uniform distribution of hash values; skew is therefore not a problem in practice.

Further, collisions, or data entries with the same hash value, cause a problem and must be handled specially: when more data entries than will fit on a page have the same hash value, we need overflow pages.
10.3 LINEAR HASHING *

Linear Hashing is a dynamic hashing technique, like Extendible Hashing, adjusting gracefully to inserts and deletes. In contrast to Extendible Hashing, it does not require a directory, deals naturally with collisions, and offers a lot of flexibility with respect to the timing of bucket splits (allowing us to trade off slightly greater overflow chains for higher average space utilization). If the data distribution is very skewed, however, overflow chains could cause Linear Hashing performance to be worse than that of Extendible Hashing.

The scheme utilizes a family of hash functions $h_0, h_1, h_2, \ldots$, with the property that each function’s range is twice that of its predecessor. That is, if $h_i$ maps a data entry into one of $M$ buckets, $h_{i+1}$ maps a data entry into one of $2M$ buckets. Such a family is typically obtained by choosing a hash function $h$ and an initial number $N$ of buckets, and defining $h_i(value) = h(value) \mod (2^i N)$. If $N$ is chosen to be a power of 2, then we apply $h$ and look at the last $d_i$ bits; $d_0$ is the number of bits needed to represent $N$, and $d_i = d_0 + i$. Typically we choose $h$ to be a function that maps a data entry to some integer. Suppose that we set the initial number $N$ of buckets to be 32. In this case $d_0 = 5$, and $h_0$ is therefore $h \mod 32$, that is, a number in the range 0 to 31. The value of $d_1$ is $d_0 + 1 = 6$, and $h_1$ is $h \mod (2 \times 32)$, that is, a number in the range 0 to 63. $h_2$ yields a number in the range 0 to 127, and so on.

The idea is best understood in terms of rounds of splitting. During round number $Level$, only hash functions $h_{Level}$ and $h_{Level+1}$ are in use. The buckets in the file at the beginning of the round are split, one by one from the first to the last bucket, thereby doubling the number of buckets. At any given point within a round, therefore, we have buckets that have been split, buckets that are yet to be split, and buckets created by splits in this round, as illustrated in Figure 10.7.

Consider how we search for a data entry with a given search key value. We apply hash function $h_{Level}$, and if this leads us to one of the unsplit buckets, we simply look there. If it leads us to one of the split buckets, the entry may be there or it may have been moved to the new bucket created earlier in this round by splitting this bucket; to determine which of these two buckets contains the entry, we apply $h_{Level+1}$.

Unlike Extendible Hashing, when an insert triggers a split, the bucket into which the data entry is inserted is not necessarily the bucket that is split. An overflow page is added to store the newly inserted data entry (which triggered the split), as in Static Hashing. However, since the bucket to split is chosen in round-robin fashion, eventually all buckets are split, thereby redistributing the data entries in overflow chains before the chains get to be more than one or two pages long.

\footnote{Note that 0 to $N-1$ is \textit{not} the range of $h$!}
We now describe Linear Hashing in more detail. A counter \( \text{Level} \) is used to indicate the current round number and is initialized to 0. The bucket to split is denoted by \( \text{Next} \) and is initially bucket 0 (the first bucket). We denote the number of buckets in the file at the beginning of round \( \text{Level} \) by \( N_{\text{Level}} \). We can easily verify that \( N_{\text{Level}} = N \times 2^{\text{Level}} \).

Let the number of buckets at the beginning of round 0, denoted by \( N_0 \), be \( N \). We show a small linear hashed file in Figure 10.8. Each bucket can hold four data entries, and the file initially contains four buckets, as shown in the figure.

We have considerable flexibility in how to trigger a split, thanks to the use of overflow pages. We can split whenever a new overflow page is added, or we can impose additional
conditions based on conditions such as space utilization. For our examples, a split is ‘triggered’ when inserting a new data entry causes the creation of an overflow page.

Whenever a split is triggered the Next bucket is split, and hash function $h_{\text{Level}+1}$ redistributes entries between this bucket (say bucket number $b$) and its split image; the split image is therefore bucket number $b + N_{\text{Level}}$. After splitting a bucket, the value of Next is incremented by 1. In the example file, insertion of data entry 43 triggers a split. The file after completing the insertion is shown in Figure 10.9.

![Figure 10.9](image_url)

**Figure 10.9** After Inserting Record $r$ with $h(r)=43$

At any time in the middle of a round Level, all buckets above bucket Next have been split, and the file contains buckets that are their split images, as illustrated in Figure 10.7. Buckets Next through $N_{\text{Level}}$ have not yet been split. If we use $h_{\text{Level}}$ on a data entry and obtain a number $b$ in the range Next through $N_{\text{Level}}$, the data entry belongs to bucket $b$. For example, $h_0(18)$ is 2 (binary 10); since this value is between the current values of Next (= 1) and $N_1$ (= 4), this bucket has not been split. However, if we obtain a number $b$ in the range 0 through Next, the data entry may be in this bucket or in its split image (which is bucket number $b + N_{\text{Level}}$); we have to use $h_{\text{Level}+1}$ to determine which of these two buckets the data entry belongs to. In other words, we have to look at one more bit of the data entry’s hash value. For example, $h_0(32)$ and $h_0(44)$ are both 0 (binary 00). Since Next is currently equal to 1, which indicates a bucket that has been split, we have to apply $h_1$. We have $h_1(32) = 0$ (binary 000) and $h_1(44) = 4$ (binary 100). Thus, 32 belongs in bucket A and 44 belongs in its split image, bucket A2.
Not all insertions trigger a split, of course. If we insert 37* into the file shown in Figure 10.9, the appropriate bucket has space for the new data entry. The file after the insertion is shown in Figure 10.10.

Sometimes the bucket pointed to by Next (the current candidate for splitting) is full, and a new data entry should be inserted in this bucket. In this case a split is triggered, of course, but we do not need a new overflow bucket. This situation is illustrated by inserting 29* into the file shown in Figure 10.10. The result is shown in Figure 10.11.

When Next is equal to \( N_{Level} - 1 \) and a split is triggered, we split the last of the buckets that were present in the file at the beginning of round Level. The number of buckets after the split is twice the number at the beginning of the round, and we start a new round with Level incremented by 1 and Next reset to 0. Incrementing Level amounts to doubling the effective range into which keys are hashed. Consider the example file in Figure 10.12, which was obtained from the file of Figure 10.11 by inserting 22*, 66*, and 34*. (The reader is encouraged to try to work out the details of these insertions.) Inserting 50* causes a split that leads to incrementing Level, as discussed above; the file after this insertion is shown in Figure 10.13.

In summary, an equality selection costs just one disk I/O unless the bucket has overflow pages; in practice, the cost on average is about 1.2 disk accesses for reasonably uniform data distributions. (The cost can be considerably worse—linear in the number of data entries in the file—if the distribution is very skewed. The space utilization is also very poor with skewed data distributions.) Inserts require reading and writing a single page, unless a split is triggered.
Figure 10.11 After Inserting Record \( r \) with \( h(r) = 29 \)

Figure 10.12 After Inserting Records with \( h(r) = 22, 66, \) and 34
We will not discuss deletion in detail, but it is essentially the inverse of insertion. If the last bucket in the file is empty, it can be removed and Next can be decremented. (If Next is 0 and the last bucket becomes empty, Next is made to point to bucket \((M/2) - 1\), where \(M\) is the current number of buckets, Level is decremented, and the empty bucket is removed.) If we wish, we can combine the last bucket with its split image even when it is not empty, using some criterion to trigger this merging, in essentially the same way. The criterion is typically based on the occupancy of the file, and merging can be done to improve space utilization.

### 10.4 EXTENDIBLE HASHING VERSUS LINEAR HASHING *

To understand the relationship between Linear Hashing and Extendible Hashing, imagine that we also have a directory in Linear Hashing with elements 0 to \(N - 1\). The first split is at bucket 0, and so we add directory element \(N\). In principle, we may imagine that the entire directory has been doubled at this point; however, because element 1 is the same as element \(N + 1\), element 2 is the same as element \(N + 2\), and so on, we can avoid the actual copying for the rest of the directory. The second split occurs at bucket 1; now directory element \(N + 1\) becomes significant and is added. At the end of the round, all the original \(N\) buckets are split, and the directory is doubled in size (because all elements point to distinct buckets).
We observe that the choice of hashing functions is actually very similar to what goes on in Extendible Hashing—in effect, moving from $h_i$ to $h_{i+1}$ in Linear Hashing corresponds to doubling the directory in Extendible Hashing. Both operations double the effective range into which key values are hashed; but whereas the directory is doubled in a single step of Extendible Hashing, moving from $h_i$ to $h_{i+1}$, along with a corresponding doubling in the number of buckets, occurs gradually over the course of a round in Linear Hashing. The new idea behind Linear Hashing is that a directory can be avoided by a clever choice of the bucket to split. On the other hand, by always splitting the appropriate bucket, Extendible Hashing may lead to a reduced number of splits and higher bucket occupancy.

The directory analogy is useful for understanding the ideas behind Extendible and Linear Hashing. However, the directory structure can be avoided for Linear Hashing (but not for Extendible Hashing) by allocating primary bucket pages consecutively, which would allow us to locate the page for bucket $i$ by a simple offset calculation. For uniform distributions, this implementation of Linear Hashing has a lower average cost for equality selections (because the directory level is eliminated). For skewed distributions, this implementation could result in any empty or nearly empty buckets, each of which is allocated at least one page, leading to poor performance relative to Extendible Hashing, which is likely to have higher bucket occupancy.

A different implementation of Linear Hashing, in which a directory is actually maintained, offers the flexibility of not allocating one page per bucket; null directory elements can be used as in Extendible Hashing. However, this implementation introduces the overhead of a directory level and could prove costly for large, uniformly distributed files. (Also, although this implementation alleviates the potential problem of low bucket occupancy by not allocating pages for empty buckets, it is not a complete solution because we can still have many pages with very few entries.)

10.5 POINTS TO REVIEW

- Hash-based indexes are designed for equality queries. A hashing function is applied to a search field value and returns a bucket number. The bucket number corresponds to a page on disk that contains all possibly relevant records. A Static Hashing index has a fixed number of primary buckets. During insertion, if the primary bucket for a data entry is full, an overflow page is allocated and linked to the primary bucket. The list of overflow pages at a bucket is called its overflow chain. Static Hashing can answer equality queries with a single disk I/O, in the absence of overflow chains. As the file grows, however, Static Hashing suffers from long overflow chains and performance deteriorates. (Section 10.1)

- Extendible Hashing is a dynamic index structure that extends Static Hashing by introducing a level of indirection in the form of a directory. Usually the size of
Hash-Based Indexing

the directory is $2^d$ for some $d$, which is called the *global depth* of the index. The correct directory entry is found by looking at the first $d$ bits of the result of the hashing function. The directory entry points to the page on disk with the actual data entries. If a page is full and a new data entry falls into that page, data entries from the full page are redistributed according to the first $l$ bits of the hashed values. The value $l$ is called the *local depth* of the page. The directory can get large if the data distribution is *skewed*. *Collisions*, which are data entries with the same hash value, have to be handled specially. *(Section 10.2)*

- *Linear Hashing* avoids a directory by splitting the buckets in a round-robin fashion. Linear Hashing proceeds in rounds. At the beginning of each round there is an initial set of buckets. Insertions can trigger bucket splits, but buckets are split sequentially in order. Overflow pages are required, but overflow chains are unlikely to be long because each bucket will be split at some point. During each round, two hash functions $h_{\text{Level}}$ and $h_{\text{Level}+1}$ are in use where $h_{\text{Level}}$ is used to locate buckets that are not yet split and $h_{\text{Level}+1}$ is used to locate buckets that already split. When all initial buckets have split, the current round ends and the next round starts. *(Section 10.3)*

- Extendible and Linear Hashing are closely related. Linear Hashing avoids a directory structure by having a predefined order of buckets to split. The disadvantage of Linear Hashing relative to Extendible Hashing is that space utilization could be lower, especially for skewed distributions, because the bucket splits are not concentrated where the data density is highest, as they are in Extendible Hashing. A directory-based implementation of Linear Hashing can improve space occupancy, but it is still likely to be inferior to Extendible Hashing in extreme cases. *(Section 10.4)*

**EXERCISES**

**Exercise 10.1** Consider the Extendible Hashing index shown in Figure 10.14. Answer the following questions about this index:

1. What can you say about the last entry that was inserted into the index?
2. What can you say about the last entry that was inserted into the index if you know that there have been no deletions from this index so far?
3. Suppose you are told that there have been no deletions from this index so far. What can you say about the last entry whose insertion into the index caused a split?
4. Show the index after inserting an entry with hash value 68.
5. Show the original index after inserting entries with hash values 17 and 69.
6. Show the original index after deleting the entry with hash value 21. (Assume that the full deletion algorithm is used.)
7. Show the original index after deleting the entry with hash value 10. Is a merge triggered by this deletion? If not, explain why. (Assume that the full deletion algorithm is used.)
Exercise 10.2 Consider the Linear Hashing index shown in Figure 10.15. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

1. What can you say about the last entry that was inserted into the index?
2. What can you say about the last entry that was inserted into the index if you know that there have been no deletions from this index so far?
3. Suppose you know that there have been no deletions from this index so far. What can you say about the last entry whose insertion into the index caused a split?

4. Show the index after inserting an entry with hash value 4.

5. Show the original index after inserting an entry with hash value 15.

6. Show the original index after deleting the entries with hash values 36 and 44. (Assume that the full deletion algorithm is used.)

7. Find a list of entries whose insertion into the original index would lead to a bucket with two overflow pages. Use as few entries as possible to accomplish this. What is the maximum number of entries that can be inserted into this bucket before a split occurs that reduces the length of this overflow chain?

**Exercise 10.3** Answer the following questions about Extendible Hashing:

1. Explain why local depth and global depth are needed.

2. After an insertion that causes the directory size to double, how many buckets have exactly one directory entry pointing to them? If an entry is then deleted from one of these buckets, what happens to the directory size? Explain your answers briefly.

3. Does Extendible Hashing guarantee at most one disk access to retrieve a record with a given key value?

4. If the hash function distributes data entries over the space of bucket numbers in a very skewed (non-uniform) way, what can you say about the size of the directory? What can you say about the space utilization in data pages (i.e., non-directory pages)?

5. Does doubling the directory require us to examine all buckets with local depth equal to global depth?

6. Why is handling duplicate key values in Extendible Hashing harder than in ISAM?

**Exercise 10.4** Answer the following questions about Linear Hashing.

1. How does Linear Hashing provide an average-case search cost of only slightly more than one disk I/O, given that overflow buckets are part of its data structure?

2. Does Linear Hashing guarantee at most one disk access to retrieve a record with a given key value?

3. If a Linear Hashing index using Alternative (1) for data entries contains $N$ records, with $P$ records per page and an average storage utilization of 80 percent, what is the worst-case cost for an equality search? Under what conditions would this cost be the actual search cost?

4. If the hash function distributes data entries over the space of bucket numbers in a very skewed (non-uniform) way, what can you say about the space utilization in data pages?

**Exercise 10.5** Give an example of when you would use each element (A or B) for each of the following ‘A versus B’ pairs:

1. A hashed index using Alternative (1) versus heap file organization.

2. Extendible Hashing versus Linear Hashing.
3. Static Hashing versus Linear Hashing.
4. Static Hashing versus ISAM.
5. Linear Hashing versus B+ trees.

Exercise 10.6 Give examples of the following:

1. A Linear Hashing index and an Extendible Hashing index with the same data entries, such that the Linear Hashing index has more pages.
2. A Linear Hashing index and an Extendible Hashing index with the same data entries, such that the Extendible Hashing index has more pages.

Exercise 10.7 Consider a relation R(a, b, c, d) containing 1,000,000 records, where each page of the relation holds 10 records. R is organized as a heap file with dense secondary indexes, and the records in R are randomly ordered. Assume that attribute a is a candidate key for R, with values lying in the range 0 to 999,999. For each of the following queries, name the approach that would most likely require the fewest I/Os for processing the query. The approaches to consider follow:

- Scanning through the whole heap file for R.
- Using a B+ tree index on attribute R.a.
- Using a hash index on attribute R.a.

The queries are:

1. Find all R tuples.
2. Find all R tuples such that a < 50.
3. Find all R tuples such that a = 50.
4. Find all R tuples such that a > 50 and a < 100.

Exercise 10.8 How would your answers to Exercise 10.7 change if attribute a is not a candidate key for R? How would they change if we assume that records in R are sorted on a?

Exercise 10.9 Consider the snapshot of the Linear Hashing index shown in Figure 10.16. Assume that a bucket split occurs whenever an overflow page is created.

1. What is the maximum number of data entries that can be inserted (given the best possible distribution of keys) before you have to split a bucket? Explain very briefly.
2. Show the file after inserting a single record whose insertion causes a bucket split.
3. (a) What is the minimum number of record insertions that will cause a split of all four buckets? Explain very briefly.
   (b) What is the value of Next after making these insertions?
   (c) What can you say about the number of pages in the fourth bucket shown after this series of record insertions?

Exercise 10.10 Consider the data entries in the Linear Hashing index for Exercise 10.9.
1. Show an Extendible Hashing index with the same data entries.

2. Answer the questions in Exercise 10.9 with respect to this index.

Exercise 10.11 In answering the following questions, assume that the full deletion algorithm is used. Assume that merging is done when a bucket becomes empty.

1. Give an example of an Extendible Hashing index in which deleting an entry reduces the global depth.

2. Give an example of a Linear Hashing index in which deleting an entry causes Next to be decremented but leaves Level unchanged. Show the file before and after the entry is deleted.

3. Give an example of a Linear Hashing index in which deleting an entry causes Level to be decremented. Show the file before and after the entry is deleted.

4. Give an example of an Extendible Hashing index and a list of entries $e_1, e_2, e_3$ such that inserting the entries in order leads to three splits and deleting them in the reverse order yields the original index. If such an example does not exist, explain.

5. Give an example of a Linear Hashing index and a list of entries $e_1, e_2, e_3$ such that inserting the entries in order leads to three splits and deleting them in the reverse order yields the original index. If such an example does not exist, explain.

PROJECT-BASED EXERCISES

Exercise 10.12 (Note to instructors: Additional details must be provided if this question is assigned. See Appendix B.) Implement Linear Hashing or Extendible Hashing in Minibase.
BIBLIOGRAPHIC NOTES

Hashing is discussed in detail in [381]. Extendible Hashing is proposed in [218]. Litwin proposed Linear Hashing in [418]. A generalization of Linear Hashing for distributed environments is described in [422].

There has been extensive research into hash-based indexing techniques. Larson describes two variations of Linear Hashing in [406] and [407]. Ramakrishna presents an analysis of hashing techniques in [529]. Hash functions that do not produce bucket overflows are studied in [530]. Order-preserving hashing techniques are discussed in [419] and [263]. Partitioned-hashing, in which each field is hashed to obtain some bits of the bucket address, extends hashing for the case of queries in which equality conditions are specified only for some of the key fields. This approach was proposed by Rivest [547] and is discussed in [656]; a further development is described in [537].
Conceptual database design gives us a set of relation schemas and integrity constraints (ICs) that can be regarded as a good starting point for the final database design. This initial design must be refined by taking the ICs into account more fully than is possible with just the ER model constructs and also by considering performance criteria and typical workloads. In this chapter we discuss how ICs can be used to refine the conceptual schema produced by translating an ER model design into a collection of relations. Workload and performance considerations are discussed in Chapter 16.

We concentrate on an important class of constraints called functional dependencies. Other kinds of ICs, for example multivalued dependencies and join dependencies, also provide useful information. They can sometimes reveal redundancies that cannot be detected using functional dependencies alone. We discuss these other constraints briefly.

This chapter is organized as follows. Section 15.1 is an overview of the schema refinement approach discussed in this chapter. We introduce functional dependencies in Section 15.2. In Section 15.3 we present several examples that highlight the problems caused by redundancy and illustrate how relational schemas obtained by translating an ER model design can nonetheless suffer from these problems. Thus, ER design is a good starting point, but we still need techniques to detect schemas with these problems and to refine such schemas to eliminate the problems. We lay the foundation for developing such schema refinement techniques in Section 15.4, where we show how to reason with functional dependency information to infer additional dependencies from a given set of dependencies.

We introduce normal forms for relations in Section 15.5; the normal form satisfied by a relation is a measure of the redundancy in the relation. A relation with redundancy can be refined by decomposing it, or replacing it with smaller relations that contain the same information, but without redundancy. We discuss decompositions and desirable properties of decompositions in Section 15.6. We show how relations can be decomposed into smaller relations that are in desirable normal forms in Section 15.7. Finally, we discuss the use of other kinds of dependencies for database design in Section 15.8.
15.1 INTRODUCTION TO SCHEMA REFINEMENT

We now present an overview of the problems that schema refinement is intended to address and a refinement approach based on decompositions. Redundant storage of information is the root cause of these problems. Although decomposition can eliminate redundancy, it can lead to problems of its own and should be used with caution.

15.1.1 Problems Caused by Redundancy

Storing the same information redundantly, that is, in more than one place within a database, can lead to several problems:

- **Redundant storage:** Some information is stored repeatedly.
- **Update anomalies:** If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.
- **Insertion anomalies:** It may not be possible to store some information unless some other information is stored as well.
- **Deletion anomalies:** It may not be possible to delete some information without losing some other information as well.

Consider a relation obtained by translating a variant of the Hourly_Emps entity set from Chapter 2:

\[
\text{Hourly\_Emps}(\text{ssn}, \text{name}, \text{lot}, \text{rating}, \text{hourly\_wages}, \text{hours\_worked})
\]

In this chapter we will omit attribute type information for brevity, since our focus is on the grouping of attributes into relations. We will often abbreviate an attribute name to a single letter and refer to a relation schema by a string of letters, one per attribute. For example, we will refer to the Hourly_Emps schema as SNLRWH (W denotes the \textit{hourly\_wages} attribute).

The key for Hourly\_Emps is \textit{ssn}. In addition, suppose that the \emph{hourly\_wages} attribute is determined by the \emph{rating} attribute. That is, for a given \emph{rating} value, there is only one permissible \emph{hourly\_wages} value. This IC is an example of a \emph{functional dependency}. It leads to possible redundancy in the relation Hourly\_Emps, as illustrated in Figure 15.1.

If the same value appears in the \textit{rating} column of two tuples, the IC tells us that the same value must appear in the \textit{hourly\_wages} column as well. This redundancy has several negative consequences:
Some information is stored multiple times. For example, the rating value 8 corresponds to the hourly wage 10, and this association is repeated three times. In addition to wasting space by storing the same information many times, redundancy leads to potential inconsistency. For example, the hourly wages in the first tuple could be updated without making a similar change in the second tuple, which is an example of an update anomaly. Also, we cannot insert a tuple for an employee unless we know the hourly wage for the employee’s rating value, which is an example of an insertion anomaly.

If we delete all tuples with a given rating value (e.g., we delete the tuples for Smethurst and Guldu) we lose the association between that rating value and its hourly wage value (a deletion anomaly).

Let us consider whether the use of null values can address some of these problems. Clearly, null values cannot help eliminate redundant storage or update anomalies. It appears that they can address insertion and deletion anomalies. For instance, to deal with the insertion anomaly example, we can insert an employee tuple with null values in the hourly wage field. However, null values cannot address all insertion anomalies. For example, we cannot record the hourly wage for a rating unless there is an employee with that rating, because we cannot store a null value in the ssn field, which is a primary key field. Similarly, to deal with the deletion anomaly example, we might consider storing a tuple with null values in all fields except rating and hourly_wages if the last tuple with a given rating would otherwise be deleted. However, this solution will not work because it requires the ssn value to be null, and primary key fields cannot be null. Thus, null values do not provide a general solution to the problems of redundancy, even though they can help in some cases. We will not discuss the use of null values further.

Ideally, we want schemas that do not permit redundancy, but at the very least we want to be able to identify schemas that do allow redundancy. Even if we choose to accept a schema with some of these drawbacks, perhaps owing to performance considerations, we want to make an informed decision.
15.1.2 Use of Decompositions

Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural. Functional dependencies (and, for that matter, other ICs) can be used to identify such situations and to suggest refinements to the schema. The essential idea is that many problems arising from redundancy can be addressed by replacing a relation with a collection of ‘smaller’ relations. Each of the smaller relations contains a (strict) subset of the attributes of the original relation. We refer to this process as decomposition of the larger relation into the smaller relations.

We can deal with the redundancy in Hourly_Emps by decomposing it into two relations:

Hourly_Emps2(ssn, name, lot, rating, hours_worked)
Wages(rating, hourly_wages)

The instances of these relations corresponding to the instance of Hourly_Emps relation in Figure 15.1 is shown in Figure 15.2.

<table>
<thead>
<tr>
<th>ssn</th>
<th>name</th>
<th>lot</th>
<th>rating</th>
<th>hours_worked</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-22-3666</td>
<td>Attishoo</td>
<td>48</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>231-31-5368</td>
<td>Smiley</td>
<td>22</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>131-24-3650</td>
<td>Smethurst</td>
<td>35</td>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>434-26-3751</td>
<td>Guldu</td>
<td>35</td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>612-67-4134</td>
<td>Madayan</td>
<td>35</td>
<td>8</td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>hourly_wages</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 15.2 Instances of Hourly_Emps2 and Wages

Note that we can easily record the hourly wage for any rating simply by adding a tuple to Wages, even if no employee with that rating appears in the current instance of Hourly_Emps. Changing the wage associated with a rating involves updating a single Wages tuple. This is more efficient than updating several tuples (as in the original design), and it also eliminates the potential for inconsistency. Notice that the insertion and deletion anomalies have also been eliminated.
15.1.3 Problems Related to Decomposition

Unless we are careful, decomposing a relation schema can create more problems than it solves. Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?
2. What problems (if any) does a given decomposition cause?

To help with the first question, several normal forms have been proposed for relations. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise. Considering the normal form of a given relation schema can help us to decide whether or not to decompose it further. If we decide that a relation schema must be decomposed further, we must choose a particular decomposition (i.e., a particular collection of smaller relations to replace the given relation).

With respect to the second question, two properties of decompositions are of particular interest. The lossless-join property enables us to recover any instance of the decomposed relation from corresponding instances of the smaller relations. The dependency-preservation property enables us to enforce any constraint on the original relation by simply enforcing some constraints on each of the smaller relations. That is, we need not perform joins of the smaller relations to check whether a constraint on the original relation is violated.

A serious drawback of decompositions is that queries over the original relation may require us to join the decomposed relations. If such queries are common, the performance penalty of decomposing the relation may not be acceptable. In this case we may choose to live with some of the problems of redundancy and not decompose the relation. It is important to be aware of the potential problems caused by such residual redundancy in the design and to take steps to avoid them (e.g., by adding some checks to application code). We will not discuss the impact of decompositions on query performance in this chapter; this issue is covered in Section 16.8.

Our goal in this chapter is to explain some powerful concepts and design guidelines based on the theory of functional dependencies. A good database designer should have a firm grasp of normal forms and what problems they (do or do not) alleviate, the technique of decomposition, and potential problems with decompositions. For example, a designer will often ask questions such as these: Is a relation in a given normal form? Is a decomposition dependency-preserving? Our objective is to explain when to raise these questions and the significance of the answers.
15.2 FUNCTIONAL DEPENDENCIES

A functional dependency (FD) is a kind of IC that generalizes the concept of a key. Let $R$ be a relation schema and let $X$ and $Y$ be nonempty sets of attributes in $R$. We say that an instance $r$ of $R$ satisfies the FD $X \rightarrow Y$\textsuperscript{1} if the following holds for every pair of tuples $t_1$ and $t_2$ in $r$:

If $t_1.X = t_2.X$, then $t_1.Y = t_2.Y$.

We use the notation $t_1.X$ to refer to the projection of tuple $t_1$ onto the attributes in $X$, in a natural extension of our TRC notation (see Chapter 4) $t.a$ for referring to attribute $a$ of tuple $t$. An FD $X \rightarrow Y$ essentially says that if two tuples agree on the values in attributes $X$, they must also agree on the values in attributes $Y$.

Figure 15.3 illustrates the meaning of the FD $AB \rightarrow C$ by showing an instance that satisfies this dependency. The first two tuples show that an FD is not the same as a key constraint: Although the FD is not violated, $AB$ is clearly not a key for the relation. The third and fourth tuples illustrate that if two tuples differ in either the $A$ field or the $B$ field, they can differ in the $C$ field without violating the FD. On the other hand, if we add a tuple $\langle a_1, b_1, c_2, d_1 \rangle$ to the instance shown in this figure, the resulting instance would violate the FD; to see this violation, compare the first tuple in the figure with the new tuple.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
</tr>
<tr>
<td>2</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d2</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
<td>b2</td>
<td>c2</td>
<td>d1</td>
</tr>
<tr>
<td>4</td>
<td>a2</td>
<td>b1</td>
<td>c3</td>
<td>d1</td>
</tr>
</tbody>
</table>

Figure 15.3 An Instance that Satisfies $AB \rightarrow C$

Recall that a legal instance of a relation must satisfy all specified ICs, including all specified FDs. As noted in Section 3.2, ICs must be identified and specified based on the semantics of the real-world enterprise being modeled. By looking at an instance of a relation, we might be able to tell that a certain FD does not hold. However, we can never deduce that an FD holds by looking at one or more instances of the relation because an FD, like other ICs, is a statement about all possible legal instances of the relation.

\textsuperscript{1}$X \rightarrow Y$ is read as $X$ functionally determines $Y$, or simply as $X$ determines $Y$. 
A primary key constraint is a special case of an FD. The attributes in the key play the role of \( X \), and the set of all attributes in the relation plays the role of \( Y \). Note, however, that the definition of an FD does not require that the set \( X \) be minimal; the additional minimality condition must be met for \( X \) to be a key. If \( X \rightarrow Y \) holds, where \( Y \) is the set of all attributes, and there is some subset \( V \) of \( X \) such that \( V \rightarrow Y \) holds, then \( X \) is a superkey; if \( V \) is a strict subset of \( X \), then \( X \) is not a key.

In the rest of this chapter, we will see several examples of FDs that are not key constraints.

### 15.3 EXAMPLES MOTIVATING SCHEMA REFINEMENT

It is natural to ask whether we need to decompose relations produced by translating an ER diagram. Shouldn’t a good ER design lead to a collection of good relations? Unfortunately, ER design can generate some schemas with redundancy problems, because it is a complex, subjective process, and certain constraints are not expressible in terms of ER diagrams. The examples in this section are intended to illustrate why decomposition of relations produced through ER design might be necessary.

#### 15.3.1 Constraints on an Entity Set

Consider the Hourly_Emps relation again. The constraint that attribute \( ssn \) is a key can be expressed as an FD:

\[
\{ssn\} \rightarrow \{ssn, name, lot, rating, hourly_wages, hours_worked\}
\]

For brevity, we will write this FD as \( S \rightarrow SNLRWH \), using a single letter to denote each attribute and omitting the set braces, but the reader should remember that both sides of an FD contain sets of attributes. In addition, the constraint that the \( hourly_wages \) attribute is determined by the \( rating \) attribute is an FD: \( R \rightarrow W \).

As we saw in Section 15.1.1, this FD led to redundant storage of rating–wage associations. *It cannot be expressed in terms of the ER model. Only FDs that determine all attributes of a relation (i.e., key constraints) can be expressed in the ER model.* Therefore, we could not detect it when we considered Hourly_Emps as an entity set during ER modeling.

We could argue that the problem with the original design was an artifact of a poor ER design, which could have been avoided by introducing an entity set called Wage_Table (with attributes \( rating \) and \( hourly_wages \)) and a relationship set Has_Wages associating Hourly_Emps and Wage_Table. The point, however, is that we could easily arrive at the original design given the subjective nature of ER modeling. Having formal techniques to identify the problem with this design, and to guide us to a better design, is very
useful. The value of such techniques cannot be underestimated when designing large schemas—schemas with more than a hundred tables are not uncommon!

15.3.2 Constraints on a Relationship Set

The previous example illustrated how FDs can help to refine the subjective decisions made during ER design, but one could argue that the best possible ER diagram would have led to the same final set of relations. Our next example shows how FD information can lead to a set of relations that eliminates some redundancy problems and is unlikely to be arrived at solely through ER design.

We revisit an example from Chapter 2. Suppose that we have entity sets Parts, Suppliers, and Departments, as well as a relationship set Contracts that involves all of them. We refer to the schema for Contracts as \( CQPSD \). A contract with contract id \( C \) specifies that a supplier \( S \) will supply some quantity \( Q \) of a part \( P \) to a department \( D \). (We have added the contract id field \( C \) to the version of the Contracts relation that was discussed in Chapter 2.)

We might have a policy that a department purchases at most one part from any given supplier. Thus, if there are several contracts between the same supplier and department, we know that the same part must be involved in all of them. This constraint is an FD, \( DS \rightarrow P \).

Again we have redundancy and its associated problems. We can address this situation by decomposing Contracts into two relations with attributes \( CQSD \) and \( SDP \). Intuitively, the relation \( SDP \) records the part supplied to a department by a supplier, and the relation \( CQSD \) records additional information about a contract. It is unlikely that we would arrive at such a design solely through ER modeling, since it is hard to formulate an entity or relationship that corresponds naturally to \( CQSD \).

15.3.3 Identifying Attributes of Entities

This example illustrates how a careful examination of FDs can lead to a better understanding of the entities and relationships underlying the relational tables; in particular, it shows that attributes can easily be associated with the ‘wrong’ entity set during ER design. The ER diagram in Figure 15.4 shows a relationship set called Works_In that is similar to the Works_In relationship set of Chapter 2, but with an additional key constraint indicating that an employee can work in at most one department. (Observe the arrow connecting Employees to Works_In.)

Using the key constraint, we can translate this ER diagram into two relations:

\[
\text{Workers}(\text{ssn, name, lot, did, since})
\]
The entity set Employees and the relationship set Works_In are mapped to a single relation, Workers. This translation is based on the second approach discussed in Section 2.4.1.

Now suppose that employees are assigned parking lots based on their department, and that all employees in a given department are assigned the same lot. This constraint is not expressible with respect to the ER diagram of Figure 15.4. It is another example of an FD: \( \text{did} \rightarrow \text{lot} \). The redundancy in this design can be eliminated by decomposing the Workers relation into two relations:

\[
\begin{align*}
\text{Workers}_2(\text{ssn, name, did, since}) \\
\text{Dept}_\text{Lots}(\text{did, lot})
\end{align*}
\]

The new design has much to recommend it. We can change the lots associated with a department by updating a single tuple in the second relation (i.e., no update anomalies). We can associate a lot with a department even if it currently has no employees, without using null values (i.e., no deletion anomalies). We can add an employee to a department by inserting a tuple to the first relation even if there is no lot associated with the employee’s department (i.e., no insertion anomalies).

Examining the two relations Departments and Dept_Lots, which have the same key, we realize that a Departments tuple and a Dept_Lots tuple with the same key value describe the same entity. This observation is reflected in the ER diagram shown in Figure 15.5.

Translating this diagram into the relational model would yield:

\[
\begin{align*}
\text{Workers}_2(\text{ssn, name, did, since}) \\
\text{Departments}(\text{did, dname, budget, lot})
\end{align*}
\]
It seems intuitive to associate lots with employees; on the other hand, the ICs reveal that in this example lots are really associated with departments. The subjective process of ER modeling could miss this point. The rigorous process of normalization would not.

### 15.3.4 Identifying Entity Sets

Consider a variant of the Reserves schema used in earlier chapters. Let Reserves contain attributes $S$, $B$, and $D$ as before, indicating that sailor $S$ has a reservation for boat $B$ on day $D$. In addition, let there be an attribute $C$ denoting the credit card to which the reservation is charged. We use this example to illustrate how FD information can be used to refine an ER design. In particular, we discuss how FD information can help to decide whether a concept should be modeled as an entity or as an attribute.

Suppose that every sailor uses a unique credit card for reservations. This constraint is expressed by the FD $S \rightarrow C$. This constraint indicates that in relation Reserves, we store the credit card number for a sailor as often as we have reservations for that sailor, and we have redundancy and potential update anomalies. A solution is to decompose Reserves into two relations with attributes $SBD$ and $SC$. Intuitively, one holds information about reservations, and the other holds information about credit cards.

It is instructive to think about an ER design that would lead to these relations. One approach is to introduce an entity set called Credit_Cards, with the sole attribute $cardno$, and a relationship set Has_Card associating Sailors and Credit_Cards. By noting that each credit card belongs to a single sailor, we can map Has_Card and Credit_Cards to a single relation with attributes $SC$. We would probably not model credit card numbers as entities if our main interest in card numbers is to indicate how a reservation is to be paid for; it suffices to use an attribute to model card numbers in this situation.
A second approach is to make \textit{cardno} an attribute of Sailors. But this approach is not very natural—a sailor may have several cards, and we are not interested in all of them. Our interest is in the one card that is used to pay for reservations, which is best modeled as an attribute of the relationship Reserves.

A helpful way to think about the design problem in this example is that we first make \textit{cardno} an attribute of Reserves and then refine the resulting tables by taking into account the FD information. (Whether we refine the design by adding \textit{cardno} to the table obtained from Sailors or by creating a new table with attributes \textit{SC} is a separate issue.)

\section{Reasoning About Functional Dependencies}

The discussion up to this point has highlighted the need for techniques that allow us to carefully examine and further refine relations obtained through ER design (or, for that matter, through other approaches to conceptual design). Before proceeding with the main task at hand, which is the discussion of such schema refinement techniques, we digress to examine FDs in more detail because they play such a central role in schema analysis and refinement.

Given a set of FDs over a relation schema $R$, there are typically several additional FDs that hold over $R$ whenever all of the given FDs hold. As an example, consider:

$\text{Workers}(ssn, \text{name, lot, did, since})$

We know that $ssn \rightarrow did$ holds, since $ssn$ is the key, and FD $did \rightarrow lot$ is given to hold. Therefore, in any legal instance of Workers, if two tuples have the same $ssn$ value, they must have the same $did$ value (from the first FD), and because they have the same $did$ value, they must also have the same $lot$ value (from the second FD). Thus, the FD $ssn \rightarrow lot$ also holds on Workers.

We say that an FD $f$ is \textbf{implied by} a given set $F$ of FDs if $f$ holds on every relation instance that satisfies all dependencies in $F$; that is, $f$ holds whenever all FDs in $F$ hold. Note that it is not sufficient for $f$ to hold on some instance that satisfies all dependencies in $F$; rather, $f$ must hold on \textit{every} instance that satisfies all dependencies in $F$.

\subsection{Closure of a Set of FDs}

The set of all FDs implied by a given set $F$ of FDs is called the \textbf{closure of $F$} and is denoted as $F^\dagger$. An important question is how we can \textbf{infer}, or compute, the closure of a given set $F$ of FDs. The answer is simple and elegant. The following three rules, called \textbf{Armstrong’s Axioms}, can be applied repeatedly to infer all FDs implied by
a set $F$ of FDs. We use $X$, $Y$, and $Z$ to denote sets of attributes over a relation schema $R$:

- **Reflexivity:** If $X \supseteq Y$, then $X \rightarrow Y$.
- **Augmentation:** If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any $Z$.
- **Transitivity:** If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Armstrong’s Axioms are **sound** in that they generate only FDs in $F^+$ when applied to a set $F$ of FDs. They are **complete** in that repeated application of these rules will generate all FDs in the closure $F^+$. (We will not prove these claims.) It is convenient to use some additional rules while reasoning about $F^+$:

- **Union:** If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.
- **Decomposition:** If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$.

These additional rules are not essential; their soundness can be proved using Armstrong’s Axioms.

To illustrate the use of these inference rules for FDs, consider a relation schema $ABC$ with FDs $A \rightarrow B$ and $B \rightarrow C$. A **trivial FD** is one in which the right side contains only attributes that also appear on the left side; such dependencies always hold due to reflexivity. Using reflexivity, we can generate all trivial dependencies, which are of the form:

$$X \rightarrow Y,$$

where $Y \subseteq X$, $X \subseteq ABC$, and $Y \subseteq ABC$.

From transitivity we get $A \rightarrow C$. From augmentation we get the nontrivial dependencies:

$$AC \rightarrow BC, AB \rightarrow AC, AB \rightarrow CB.$$

As a second example, we use a more elaborate version of the Contracts relation:

Contracts($contractid, supplierid, projectid, deptid, partid, qty, value$)

We denote the schema for Contracts as $CSJDPQV$. The meaning of a tuple in this relation is that the contract with $contractid C$ is an agreement that supplier $S$ ($supplierid$) will supply $Q$ items of part $P$ ($partid$) to project $J$ ($projectid$) associated with department $D$ ($deptid$); the value $V$ of this contract is equal to $value$.

The following ICs are known to hold:
1. The contract id $C$ is a key: $C \rightarrow CSJDPQV$.

2. A project purchases a given part using a single contract: $JP \rightarrow C$.

3. A department purchases at most one part from a supplier: $SD \rightarrow P$.

Several additional FDs hold in the closure of the set of given FDs:

From $JP \rightarrow C$, $C \rightarrow CSJDPQV$ and transitivity, we infer $JP \rightarrow CSJDPQV$.

From $SD \rightarrow P$ and augmentation, we infer $SDJ \rightarrow JP$.

From $SDJ \rightarrow JP$, $JP \rightarrow CSJDPQV$ and transitivity, we infer $SDJ \rightarrow CSJDPQV$. (Incidentally, while it may appear tempting to do so, we cannot conclude $SD \rightarrow CSDPQV$, canceling $J$ on both sides. FD inference is not like arithmetic multiplication!)

We can infer several additional FDs that are in the closure by using augmentation or decomposition. For example, from $C \rightarrow CSJDPQV$, using decomposition we can infer:

$$C \rightarrow C, \ C \rightarrow S, \ C \rightarrow J, \ C \rightarrow D,$$ etc.

Finally, we have a number of trivial FDs from the reflexivity rule.

### 15.4.2 Attribute Closure

If we just want to check whether a given dependency, say, $X \rightarrow Y$, is in the closure of a set $F$ of FDs, we can do so efficiently without computing $F^+$. We first compute the attribute closure $X^+$ with respect to $F$, which is the set of attributes $A$ such that $X \rightarrow A$ can be inferred using the Armstrong Axioms. The algorithm for computing the attribute closure of a set $X$ of attributes is shown in Figure 15.6.

\[
\text{closure} = X; \\
\text{repeat until there is no change:} \{ \\
\text{if there is an FD } U \rightarrow V \text{ in } F \text{ such that } U \subseteq \text{closure}, \\
\text{then set } \text{closure} = \text{closure} \cup V \\
\}\]

**Figure 15.6** Computing the Attribute Closure of Attribute Set $X$

This algorithm can be modified to find keys by starting with set $X$ containing a single attribute and stopping as soon as $\text{closure}$ contains all attributes in the relation schema.
By varying the starting attribute and the order in which the algorithm considers FDs, we can obtain all candidate keys.

15.5 NORMAL FORMS

Given a relation schema, we need to decide whether it is a good design or whether we need to decompose it into smaller relations. Such a decision must be guided by an understanding of what problems, if any, arise from the current schema. To provide such guidance, several normal forms have been proposed. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise.

The normal forms based on FDs are first normal form (1NF), second normal form (2NF), third normal form (3NF), and Boyce-Codd normal form (BCNF). These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF, every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF. A relation is in first normal form if every field contains only atomic values, that is, not lists or sets. This requirement is implicit in our definition of the relational model. Although some of the newer database systems are relaxing this requirement, in this chapter we will assume that it always holds. 2NF is mainly of historical interest. 3NF and BCNF are important from a database design standpoint.

While studying normal forms, it is important to appreciate the role played by FDs. Consider a relation schema $R$ with attributes $ABC$. In the absence of any ICs, any set of ternary tuples is a legal instance and there is no potential for redundancy. On the other hand, suppose that we have the FD $A \rightarrow B$. Now if several tuples have the same $A$ value, they must also have the same $B$ value. This potential redundancy can be predicted using the FD information. If more detailed ICs are specified, we may be able to detect more subtle redundancies as well.

We will primarily discuss redundancy that is revealed by FD information. In Section 15.8, we discuss more sophisticated ICs called multivalued dependencies and join dependencies and normal forms based on them.

15.5.1 Boyce-Codd Normal Form

Let $R$ be a relation schema, $X$ be a subset of the attributes of $R$, and let $A$ be an attribute of $R$. $R$ is in Boyce-Codd normal form if for every FD $X \rightarrow A$ that holds over $R$, one of the following statements is true:

- $A \in X$; that is, it is a trivial FD, or
- $X$ is a superkey.
Note that if we are given a set \( F \) of FDs, according to this definition, we must consider each dependency \( X \rightarrow A \) in the closure \( F^+ \) to determine whether \( R \) is in BCNF. However, we can prove that it is sufficient to check whether the left side of each dependency in \( F \) is a superkey (by computing the attribute closure and seeing if it includes all attributes of \( R \)).

Intuitively, in a BCNF relation the only nontrivial dependencies are those in which a key determines some attribute(s). Thus, each tuple can be thought of as an entity or relationship, identified by a key and described by the remaining attributes. Kent puts this colorfully, if a little loosely: “Each attribute must describe [an entity or relationship identified by] the key, the whole key, and nothing but the key.” If we use ovals to denote attributes or sets of attributes and draw arcs to indicate FDs, a relation in BCNF has the structure illustrated in Figure 15.7, considering just one key for simplicity. (If there are several candidate keys, each candidate key can play the role of KEY in the figure, with the other attributes being the ones not in the chosen candidate key.)

![Figure 15.7 FDs in a BCNF Relation](image)

BCNF ensures that no redundancy can be detected using FD information alone. It is thus the most desirable normal form (from the point of view of redundancy) if we take into account only FD information. This point is illustrated in Figure 15.8.

![Figure 15.8 Instance Illustrating BCNF](image)

This figure shows (two tuples in) an instance of a relation with three attributes \( X \), \( Y \), and \( A \). There are two tuples with the same value in the \( X \) column. Now suppose that we know that this instance satisfies an FD \( X \rightarrow A \). We can see that one of the tuples has the value \( a \) in the \( A \) column. What can we infer about the value in the \( A \) column in the second tuple? Using the FD, we can conclude that the second tuple also has the value \( a \) in this column. (Note that this is really the only kind of inference we can make about values in the fields of tuples by using FDs.)

But isn’t this situation an example of redundancy? We appear to have stored the value \( a \) twice. Can such a situation arise in a BCNF relation? No! If this relation is
in BCNF, because $A$ is distinct from $X$ it follows that $X$ must be a key. (Otherwise, the FD $X \rightarrow A$ would violate BCNF.) If $X$ is a key, then $y_1 = y_2$, which means that the two tuples are identical. Since a relation is defined to be a set of tuples, we cannot have two copies of the same tuple and the situation shown in Figure 15.8 cannot arise.

Thus, if a relation is in BCNF, every field of every tuple records a piece of information that cannot be inferred (using only FDs) from the values in all other fields in (all tuples of) the relation instance.

15.5.2 Third Normal Form

Let $R$ be a relation schema, $X$ be a subset of the attributes of $R$, and $A$ be an attribute of $R$. $R$ is in third normal form if for every FD $X \rightarrow A$ that holds over $R$, one of the following statements is true:

- $A \in X$; that is, it is a trivial FD, or
- $X$ is a superkey, or
- $A$ is part of some key for $R$.

The definition of 3NF is similar to that of BCNF, with the only difference being the third condition. Every BCNF relation is also in 3NF. To understand the third condition, recall that a key for a relation is a minimal set of attributes that uniquely determines all other attributes. $A$ must be part of a key (any key, if there are several). It is not enough for $A$ to be part of a superkey, because the latter condition is satisfied by each and every attribute! Finding all keys of a relation schema is known to be an NP-complete problem, and so is the problem of determining whether a relation schema is in 3NF.

Suppose that a dependency $X \rightarrow A$ causes a violation of 3NF. There are two cases:

- $X$ is a proper subset of some key $K$. Such a dependency is sometimes called a partial dependency. In this case we store $(X, A)$ pairs redundantly. As an example, consider the Reserves relation with attributes $SBDC$ from Section 15.3.4. The only key is $SBD$, and we have the FD $S \rightarrow C$. We store the credit card number for a sailor as many times as there are reservations for that sailor.

- $X$ is not a proper subset of any key. Such a dependency is sometimes called a transitive dependency because it means we have a chain of dependencies $K \rightarrow X \rightarrow A$. The problem is that we cannot associate an $X$ value with a $K$ value unless we also associate an $A$ value with an $X$ value. As an example, consider the Hourly_Emps relation with attributes $SNLRWH$ from Section 15.3.1. The only key is $S$, but there is an FD $R \rightarrow W$, which gives rise to the chain $S \rightarrow R$
→ W. The consequence is that we cannot record the fact that employee $S$ has rating $R$ without knowing the hourly wage for that rating. This condition leads to insertion, deletion, and update anomalies.

Partial dependencies are illustrated in Figure 15.9, and transitive dependencies are illustrated in Figure 15.10. Note that in Figure 15.10, the set $X$ of attributes may or may not have some attributes in common with KEY; the diagram should be interpreted as indicating only that $X$ is not a subset of KEY.

The motivation for 3NF is rather technical. By making an exception for certain dependencies involving key attributes, we can ensure that every relation schema can be decomposed into a collection of 3NF relations using only decompositions that have certain desirable properties (Section 15.6). Such a guarantee does not exist for BCNF relations; the 3NF definition weakens the BCNF requirements just enough to make this guarantee possible. We may therefore compromise by settling for a 3NF design. As we shall see in Chapter 16, we may sometimes accept this compromise (or even settle for a non-3NF schema) for other reasons as well.

Unlike BCNF, however, some redundancy is possible with 3NF. The problems associated with partial and transitive dependencies persist if there is a nontrivial dependency $X \rightarrow A$ and $X$ is not a superkey, even if the relation is in 3NF because $A$ is part of a key. To understand this point, let us revisit the Reserves relation with attributes $SBDC$ and the FD $S \rightarrow C$, which states that a sailor uses a unique credit card to pay for reservations. $S$ is not a key, and $C$ is not part of a key. (In fact, the only key is $SBD$.) Thus, this relation is not in 3NF; $(S, C)$ pairs are stored redundantly. However, if we also know that credit cards uniquely identify the owner, we have the FD $C \rightarrow$
S, which means that $CBD$ is also a key for Reserves. Therefore, the dependency $S \rightarrow C$ does not violate 3NF, and Reserves is in 3NF. Nonetheless, in all tuples containing the same $S$ value, the same $(S, C)$ pair is redundantly recorded.

For completeness, we remark that the definition of **second normal form** is essentially that partial dependencies are not allowed. Thus, if a relation is in 3NF (which precludes both partial and transitive dependencies), it is also in 2NF.

### 15.6 DECOMPOSITIONS

As we have seen, a relation in BCNF is free of redundancy (to be precise, redundancy that can be detected using FD information), and a relation schema in 3NF comes close. If a relation schema is not in one of these normal forms, the FDs that cause a violation can give us insight into the potential problems. The main technique for addressing such redundancy-related problems is decomposing a relation schema into relation schemas with fewer attributes.

A decomposition of a relation schema $R$ consists of replacing the relation schema by two (or more) relation schemas that each contain a subset of the attributes of $R$ and together include all attributes in $R$. Intuitively, we want to store the information in any given instance of $R$ by storing projections of the instance. This section examines the use of decompositions through several examples.

We begin with the Hourly_Emps example from Section 15.3.1. This relation has attributes $SNLRWH$ and two FDs: $S \rightarrow SNLRWH$ and $R \rightarrow W$. Since $R$ is not a key and $W$ is not part of any key, the second dependency causes a violation of 3NF.

The alternative design consisted of replacing Hourly_Emps with two relations having attributes $SNLRH$ and $RW$. $S \rightarrow SNLRH$ holds over $SNLRH$, and $S$ is a key. $R \rightarrow W$ holds over $RW$, and $R$ is a key for $RW$. The only other dependencies that hold over these schemas are those obtained by augmentation. Thus both schemas are in BCNF.

Our decision to decompose $SNLRWH$ into $SNLRH$ and $RW$, rather than, say, $SNLR$ and $LRWH$, was not just a good guess. It was guided by the observation that the dependency $R \rightarrow W$ caused the violation of 3NF; the most natural way to deal with this violation is to remove the attribute $W$ from this schema. To compensate for removing $W$ from the main schema, we can add a relation $RW$, because each $R$ value is associated with at most one $W$ value according to the FD $R \rightarrow W$.

A very important question must be asked at this point: If we replace a legal instance $r$ of relation schema $SNLRWH$ with its projections on $SNLRH$ ($r_1$) and $RW$ ($r_2$), can we recover $r$ from $r_1$ and $r_2$? The decision to decompose $SNLRWH$ into $SNLRH$ and $RW$ is equivalent to saying that we will store instances $r_1$ and $r_2$ instead of $r$. However,
it is the instance \( r \) that captures the intended entities or relationships. If we cannot compute \( r \) from \( r_1 \) and \( r_2 \), our attempt to deal with redundancy has effectively thrown out the baby with the bathwater. We consider this issue in more detail below.

### 15.6.1 Lossless-Join Decomposition

Let \( R \) be a relation schema and let \( F \) be a set of FDs over \( R \). A decomposition of \( R \) into two schemas with attribute sets \( X \) and \( Y \) is said to be a **lossless-join decomposition with respect to \( F \)** if for every instance \( r \) of \( R \) that satisfies the dependencies in \( F \),

\[
\pi_X(r) \Join \pi_Y(r) = r.
\]

This definition can easily be extended to cover a decomposition of \( R \) into more than two relations. It is easy to see that \( r \subseteq \pi_X(r) \Join \pi_Y(r) \) always holds. In general, though, the other direction does not hold. If we take projections of a relation and recombine them using natural join, we typically obtain some tuples that were not in the original relation. This situation is illustrated in Figure 15.11.

![Figure 15.11](image)

By replacing the instance \( r \) shown in Figure 15.11 with the instances \( \pi_{SP}(r) \) and \( \pi_{PD}(r) \), we lose some information. In particular, suppose that the tuples in \( r \) denote relationships. We can no longer tell that the relationships \((s_1, p_1, d_3)\) and \((s_3, p_1, d_1)\) do not hold. The decomposition of schema \( SPD \) into \( SP \) and \( PD \) is therefore a ‘lossy’ decomposition if the instance \( r \) shown in the figure is legal, that is, if this instance could arise in the enterprise being modeled. (Observe the similarities between this example and the Contracts relationship set in Section 2.5.3.)

**All decompositions used to eliminate redundancy must be lossless.** The following simple test is very useful:

Let \( R \) be a relation and \( F \) be a set of FDs that hold over \( R \). The decomposition of \( R \) into relations with attribute sets \( R_1 \) and \( R_2 \) is lossless if and only if \( F^+ \) contains either the FD \( R_1 \cap R_2 \rightarrow R_1 \) or the FD \( R_1 \cap R_2 \rightarrow R_2 \).
In other words, the attributes common to $R_1$ and $R_2$ must contain a key for either $R_1$ or $R_2$. If a relation is decomposed into two relations, this test is a necessary and sufficient condition for the decomposition to be lossless-join.\(^2\) If a relation is decomposed into more than two relations, an efficient (time polynomial in the size of the dependency set) algorithm is available to test whether or not the decomposition is lossless, but we will not discuss it.

Consider the Hourly_Emps relation again. It has attributes $SNLRWH$, and the FD $R \rightarrow W$ causes a violation of 3NF. We dealt with this violation by decomposing the relation into $SNLRH$ and $RW$. Since $R$ is common to both decomposed relations, and $R \rightarrow W$ holds, this decomposition is lossless-join.

This example illustrates a general observation:

If an FD $X \rightarrow Y$ holds over a relation $R$ and $X \cap Y$ is empty, the decomposition of $R$ into $R - Y$ and $XY$ is lossless.

$X$ appears in both $R - Y$ (since $X \cap Y$ is empty) and $XY$, and it is a key for $XY$. Thus, the above observation follows from the test for a lossless-join decomposition.

Another important observation has to do with repeated decompositions. Suppose that a relation $R$ is decomposed into $R_1$ and $R_2$ through a lossless-join decomposition, and that $R_1$ is decomposed into $R_{11}$ and $R_{12}$ through another lossless-join decomposition. Then the decomposition of $R$ into $R_{11}$, $R_{12}$, and $R_2$ is lossless-join; by joining $R_{11}$ and $R_{12}$ we can recover $R_1$, and by then joining $R_1$ and $R_2$, we can recover $R$.

### 15.6.2 Dependency-Preserving Decomposition

Consider the Contracts relation with attributes $CSJDPQV$ from Section 15.4.1. The given FDs are $C \rightarrow CSJDPQV$, $JP \rightarrow C$, and $SD \rightarrow P$. Because $SD$ is not a key the dependency $SD \rightarrow P$ causes a violation of BCNF.

We can decompose Contracts into two relations with schemas $CSJDQV$ and $SDP$ to address this violation; the decomposition is lossless-join. There is one subtle problem, however. We can enforce the integrity constraint $JP \rightarrow C$ easily when a tuple is inserted into Contracts by ensuring that no existing tuple has the same $JP$ values (as the inserted tuple) but different $C$ values. Once we decompose Contracts into $CSJDQV$ and $SDP$, enforcing this constraint requires an expensive join of the two relations whenever a tuple is inserted into $CSJDQV$. We say that this decomposition is not dependency-preserving.

\(^2\)It is necessary only if we assume that only functional dependencies can be specified as integrity constraints. See Exercise 15.8.
Intuitively, a *dependency-preserving decomposition* allows us to enforce all FDs by examining a single relation instance on each insertion or modification of a tuple. (Note that deletions cannot cause violation of FDs.) To define dependency-preserving decompositions precisely, we have to introduce the concept of a projection of FDs.

Let $R$ be a relation schema that is decomposed into two schemas with attribute sets $X$ and $Y$, and let $F$ be a set of FDs over $R$. The *projection of $F$ on $X$* is the set of FDs in the closure $F^+$ (not just $F$!) that involve only attributes in $X$. We will denote the projection of $F$ on attributes $X$ as $F^X$. Note that a dependency $U \rightarrow V$ in $F^+$ is in $F^X$ only if all the attributes in $U$ and $V$ are in $X$.

The decomposition of relation schema $R$ with FDs $F$ into schemas with attribute sets $X$ and $Y$ is *dependency-preserving* if $(F^X \cup F^Y)^+ = F^+$. That is, if we take the dependencies in $F^X$ and $F^Y$ and compute the closure of their union, we get back all dependencies in the closure of $F$. Therefore, we need to enforce only the dependencies in $F^X$ and $F^Y$; all FDs in $F^+$ are then sure to be satisfied. To enforce $F^X$, we need to examine only relation $X$ (on inserts to that relation). To enforce $F^Y$, we need to examine only relation $Y$.

To appreciate the need to consider the closure $F^+$ while computing the projection of $F$, suppose that a relation $R$ with attributes $ABC$ is decomposed into relations with attributes $AB$ and $BC$. The set $F$ of FDs over $R$ includes $A \rightarrow B$, $B \rightarrow C$, and $C \rightarrow A$. Of these, $A \rightarrow B$ is in $F_{AB}$ and $B \rightarrow C$ is in $F_{BC}$. But is this decomposition dependency-preserving? What about $C \rightarrow A$? This dependency is not implied by the dependencies listed (thus far) for $F_{AB}$ and $F_{BC}$.

The closure of $F$ contains all dependencies in $F$ plus $A \rightarrow C$, $B \rightarrow A$, and $C \rightarrow B$. Consequently, $F_{AB}$ also contains $B \rightarrow A$, and $F_{BC}$ contains $C \rightarrow B$. Thus, $F_{AB} \cup F_{BC}$ contains $A \rightarrow B$, $B \rightarrow C$, $B \rightarrow A$, and $C \rightarrow B$. The closure of the dependencies in $F_{AB}$ and $F_{BC}$ now includes $C \rightarrow A$ (which follows from $C \rightarrow B$, $B \rightarrow A$, and transitivity). Thus, the decomposition preserves the dependency $C \rightarrow A$.

A direct application of the definition gives us a straightforward algorithm for testing whether a decomposition is dependency-preserving. (This algorithm is exponential in the size of the dependency set; a polynomial algorithm is available, although we will not discuss it.)

We began this section with an example of a lossless-join decomposition that was not dependency-preserving. Other decompositions are dependency-preserving, but not lossless. A simple example consists of a relation $ABC$ with FD $A \rightarrow B$ that is decomposed into $AB$ and $BC$. 


15.7 NORMALIZATION

Having covered the concepts needed to understand the role of normal forms and decompositions in database design, we now consider algorithms for converting relations to BCNF or 3NF. If a relation schema is not in BCNF, it is possible to obtain a lossless-join decomposition into a collection of BCNF relation schemas. Unfortunately, there may not be any dependency-preserving decomposition into a collection of BCNF relation schemas. However, there is always a dependency-preserving, lossless-join decomposition into a collection of 3NF relation schemas.

15.7.1 Decomposition into BCNF

We now present an algorithm for decomposing a relation schema $R$ into a collection of BCNF relation schemas:

1. Suppose that $R$ is not in BCNF. Let $X \subseteq R$, $A$ be a single attribute in $R$, and $X \rightarrow A$ be an FD that causes a violation of BCNF. Decompose $R$ into $R - A$ and $XA$.

2. If either $R - A$ or $XA$ is not in BCNF, decompose them further by a recursive application of this algorithm.

$R - A$ denotes the set of attributes other than $A$ in $R$, and $XA$ denotes the union of attributes in $X$ and $A$. Since $X \rightarrow A$ violates BCNF, it is not a trivial dependency; further, $A$ is a single attribute. Therefore, $A$ is not in $X$; that is, $X \cap A$ is empty. Thus, each decomposition carried out in Step (1) is lossless-join.

The set of dependencies associated with $R - A$ and $XA$ is the projection of $F$ onto their attributes. If one of the new relations is not in BCNF, we decompose it further in Step (2). Since a decomposition results in relations with strictly fewer attributes, this process will terminate, leaving us with a collection of relation schemas that are all in BCNF. Further, joining instances of the (two or more) relations obtained through this algorithm will yield precisely the corresponding instance of the original relation (i.e., the decomposition into a collection of relations that are each in BCNF is a lossless-join decomposition).

Consider the Contracts relation with attributes $CSJDPQV$ and key $C$. We are given FDs $JP \rightarrow C$ and $SD \rightarrow P$. By using the dependency $SD \rightarrow P$ to guide the decomposition, we get the two schemas $SDP$ and $CSJDQV$. $SDP$ is in BCNF. Suppose that we also have the constraint that each project deals with a single supplier: $J \rightarrow S$. This means that the schema $CSJDQV$ is not in BCNF. So we decompose it further into $JS$ and $CJDQV$. $C \rightarrow JDQV$ holds over $CJDQV$; the only other FDs that hold are those obtained from this FD by augmentation, and therefore all FDs contain a key in the left
side. Thus, each of the schemas $SDP$, $JS$, and $CJDQV$ is in BCNF, and this collection of schemas also represents a lossless-join decomposition of $CSJDQV$.

The steps in this decomposition process can be visualized as a tree, as shown in Figure 15.12. The root is the original relation $CSJDPQV$, and the leaves are the BCNF relations that are the result of the decomposition algorithm, namely, $SDP$, $JS$, and $CSDQV$. Intuitively, each internal node is replaced by its children through a single decomposition step that is guided by the FD shown just below the node.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{decomposition_tree.png}
\caption{Decomposition of $CSJDQV$ into $SDP$, $JS$, and $CJDQV$}
\end{figure}

**Redundancy in BCNF Revisited**

The decomposition of $CSJDQV$ into $SDP$, $JS$, and $CJDQV$ is not dependency-preserving. Intuitively, dependency $JP \rightarrow C$ cannot be enforced without a join. One way to deal with this situation is to add a relation with attributes $CJP$. In effect, this solution amounts to storing some information redundantly in order to make the dependency enforcement cheaper.

This is a subtle point: Each of the schemas $CJP$, $SDP$, $JS$, and $CJDQV$ is in BCNF, yet there is some redundancy that can be predicted by FD information. In particular, if we join the relation instances for $SDP$ and $CJDQV$ and project onto the attributes $CJP$, we must get exactly the instance stored in the relation with schema $CJP$. We saw in Section 15.5.1 that there is no such redundancy within a single BCNF relation. The current example shows that redundancy can still occur across relations, even though there is no redundancy within a relation.
Alternatives in Decomposing to BCNF

Suppose that several dependencies violate BCNF. Depending on which of these dependencies we choose to guide the next decomposition step, we may arrive at quite different collections of BCNF relations. Consider Contracts. We just decomposed it into $SDP$, $JS$, and $CJDQV$. Suppose that we choose to decompose the original relation $CSJDPQV$ into $JS$ and $CJDQPQV$, based on the FD $J \rightarrow S$. The only dependencies that hold over $CJDQPQV$ are $JP \rightarrow C$ and the key dependency $C \rightarrow CJDQPQV$. Since $JP$ is a key, $CJDQPQV$ is in BCNF. Thus, the schemas $JS$ and $CJDQPQV$ represent a lossless-join decomposition of Contracts into BCNF relations.

The lesson to be learned here is that the theory of dependencies can tell us when there is redundancy and give us clues about possible decompositions to address the problem, but it cannot discriminate between decomposition alternatives. A designer has to consider the alternatives and choose one based on the semantics of the application.

BCNF and Dependency-Preservation

Sometimes, there simply is no decomposition into BCNF that is dependency-preserving. As an example, consider the relation schema $SBD$, in which a tuple denotes that sailor $S$ has reserved boat $B$ on date $D$. If we have the FDs $SB \rightarrow D$ (a sailor can reserve a given boat for at most one day) and $D \rightarrow B$ (on any given day at most one boat can be reserved), $SBD$ is not in BCNF because $D$ is not a key. If we try to decompose it, however, we cannot preserve the dependency $SB \rightarrow D$.

15.7.2 Decomposition into 3NF *

Clearly, the approach that we outlined for lossless-join decomposition into BCNF will also give us a lossless-join decomposition into 3NF. (Typically, we can stop a little earlier if we are satisfied with a collection of 3NF relations.) But this approach does not ensure dependency-preservation.

A simple modification, however, yields a decomposition into 3NF relations that is lossless-join and dependency-preserving. Before we describe this modification, we need to introduce the concept of a minimal cover for a set of FDs.

Minimal Cover for a Set of FDs

A minimal cover for a set $F$ of FDs is a set $G$ of FDs such that:

1. Every dependency in $G$ is of the form $X \rightarrow A$, where $A$ is a single attribute.
2. The closure $F^+$ is equal to the closure $G^+$.
3. If we obtain a set $H$ of dependencies from $G$ by deleting one or more dependencies, or by deleting attributes from a dependency in $G$, then $F^+ \neq H^+$.

Intuitively, a minimal cover for a set $F$ of FDs is an equivalent set of dependencies that is *minimal* in two respects: (1) Every dependency is as small as possible; that is, each attribute on the left side is necessary and the right side is a single attribute. (2) Every dependency in it is required in order for the closure to be equal to $F^+$.

As an example, let $F$ be the set of dependencies:

\[ A \rightarrow B, \; ABCD \rightarrow E, \; EF \rightarrow G, \; EF \rightarrow H, \; \text{and} \; ACDF \rightarrow EG. \]

First, let us rewrite $ACDF \rightarrow EG$ so that every right side is a single attribute:

\[ ACDF \rightarrow E \text{ and } ACDF \rightarrow G. \]

Next consider $ACDF \rightarrow G$. This dependency is implied by the following FDs:

\[ A \rightarrow B, \; ABCD \rightarrow E, \; \text{and} \; EF \rightarrow G. \]

Therefore, we can delete it. Similarly, we can delete $ACDF \rightarrow E$. Next consider $ABCD \rightarrow E$. Since $A \rightarrow B$ holds, we can replace it with $ACD \rightarrow E$. (At this point, the reader should verify that each remaining FD is minimal and required.) Thus, a minimal cover for $F$ is the set:

\[ A \rightarrow B, \; ACD \rightarrow E, \; EF \rightarrow G, \; \text{and} \; EF \rightarrow H. \]

The preceding example suggests a general algorithm for obtaining a minimal cover of a set $F$ of FDs:

1. **Put the FDs in a standard form:** Obtain a collection $G$ of equivalent FDs with a single attribute on the right side (using the decomposition axiom).

2. **Minimize the left side of each FD:** For each FD in $G$, check each attribute in the left side to see if it can be deleted while preserving equivalence to $F^+$.

3. **Delete redundant FDs:** Check each remaining FD in $G$ to see if it can be deleted while preserving equivalence to $F^+$.

Note that the order in which we consider FDs while applying these steps could produce different minimal covers; there could be several minimal covers for a given set of FDs.

More important, it is necessary to minimize the left sides of FDs before checking for redundant FDs. If these two steps are reversed, the final set of FDs could still contain some redundant FDs (i.e., not be a minimal cover), as the following example illustrates. Let $F$ be the set of dependencies, each of which is already in the standard form:
\[ A B C D \rightarrow E, E \rightarrow D, A \rightarrow B, \text{ and } A C \rightarrow D. \]

Observe that none of these FDs is redundant; if we checked for redundant FDs first, we would get the same set of FDs \( F \). The left side of \( A B C D \rightarrow E \) can be replaced by \( A C \) while preserving equivalence to \( F^+ \), and we would stop here if we checked for redundant FDs in \( F \) before minimizing the left sides. However, the set of FDs we have is not a minimal cover:

\[ A C \rightarrow E, E \rightarrow D, A \rightarrow B, \text{ and } A C \rightarrow D. \]

From transitivity, the first two FDs imply the last FD, which can therefore be deleted while preserving equivalence to \( F^+ \). The important point to note is that \( A C \rightarrow D \) becomes redundant only after we replace \( A B C D \rightarrow E \) with \( A C \rightarrow E \). If we minimize left sides of FDs first and then check for redundant FDs, we are left with the first three FDs in the preceding list, which is indeed a minimal cover for \( F \).

**Dependency-Preserving Decomposition into 3NF**

Returning to the problem of obtaining a lossless-join, dependency-preserving decomposition into 3NF relations, let \( R \) be a relation with a set \( F \) of FDs that is a minimal cover, and let \( R_1, R_2, \ldots, R_n \) be a lossless-join decomposition of \( R \). For \( 1 \leq i \leq n \), suppose that each \( R_i \) is in 3NF and let \( F_i \) denote the projection of \( F \) onto the attributes of \( R_i \). Do the following:

- Identify the set \( N \) of dependencies in \( F \) that are not preserved, that is, not included in the closure of the union of \( F_i \)s.
- For each FD \( X \rightarrow A \) in \( N \), create a relation schema \( XA \) and add it to the decomposition of \( R \).

Obviously, every dependency in \( F \) is preserved if we replace \( R \) by the \( R_i \)s plus the schemas of the form \( XA \) added in this step. The \( R_i \)s are given to be in 3NF. We can show that each of the schemas \( XA \) is in 3NF as follows: Since \( X \rightarrow A \) is in the minimal cover \( F \), \( Y \rightarrow A \) does not hold for any \( Y \) that is a strict subset of \( X \). Therefore, \( X \) is a key for \( XA \). Further, if any other dependencies hold over \( XA \), the right side can involve only attributes in \( X \) because \( A \) is a single attribute (because \( X \rightarrow A \) is an FD in a minimal cover). Since \( X \) is a key for \( XA \), none of these additional dependencies causes a violation of 3NF (although they might cause a violation of BCNF).

As an optimization, if the set \( N \) contains several FDs with the same left side, say, \( X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n \), we can replace them with a single equivalent FD \( X \rightarrow A_1 \ldots A_n \). Therefore, we produce one relation schema \( XA_1 \ldots A_n \), instead of several schemas \( XA_1, \ldots, XA_n \), which is generally preferable.
Consider the Contracts relation with attributes \(CSJDPQV\) and FDs \(JP \rightarrow C, SD \rightarrow P,\) and \(J \rightarrow S.\) If we decompose \(CSJDPQV\) into \(SDP\) and \(CSJDQV,\) then \(SDP\) is in BCNF, but \(CSJDQV\) is not even in 3NF. So we decompose it further into \(JS\) and \(CJDQV.\) The relation schemas \(SDP, JS,\) and \(CJDQV\) are in 3NF (in fact, in BCNF), and the decomposition is lossless-join. However, the dependency \(JP \rightarrow C\) is not preserved. This problem can be addressed by adding a relation schema \(CJP\) to the decomposition.

### 3NF Synthesis

We have assumed that the design process starts with an ER diagram, and that our use of FDs is primarily to guide decisions about decomposition. The algorithm for obtaining a lossless-join, dependency-preserving decomposition was presented in the previous section from this perspective—a lossless-join decomposition into 3NF is straightforward, and the algorithm addresses dependency-preservation by adding extra relation schemas.

An alternative approach, called synthesis, is to take all the attributes over the original relation \(R\) and a minimal cover \(F\) for the FDs that hold over it, and to add a relation schema \(XA\) to the decomposition of \(R\) for each FD \(X \rightarrow A\) in \(F.\)

The resulting collection of relation schemas is in 3NF and preserves all FDs. If it is not a lossless-join decomposition of \(R,\) we can make it so by adding a relation schema that contains just those attributes that appear in some key. This algorithm gives us a lossless-join, dependency-preserving decomposition into 3NF, and has polynomial complexity—polynomial algorithms are available for computing minimal covers, and a key can be found in polynomial time (even though finding all keys is known to be NP-complete). The existence of a polynomial algorithm for obtaining a lossless-join, dependency-preserving decomposition into 3NF is surprising when we consider that testing whether a given schema is in 3NF is NP-complete.

As an example, consider a relation \(ABC\) with FDs \(F = \{A \rightarrow B, C \rightarrow B\}.\) The first step yields the relation schemas \(AB\) and \(BC.\) This is not a lossless-join decomposition of \(ABC; AB \cap BC\) is \(B,\) and neither \(B \rightarrow A\) nor \(B \rightarrow C\) is in \(F^+.\) If we add a schema \(AC,\) we have the lossless-join property as well. Although the collection of relations \(AB, BC,\) and \(AC\) is a dependency-preserving, lossless-join decomposition of \(ABC,\) we obtained it through a process of synthesis, rather than through a process of repeated decomposition. We note that the decomposition produced by the synthesis approach is heavily dependent on the minimal cover that is used.

As another example of the synthesis approach, consider the Contracts relation with attributes \(CSJDPQV\) and the following FDs:

\[C \rightarrow CSJDPQV, JP \rightarrow C, SD \rightarrow P,\text{ and } J \rightarrow S.\]
This set of FDs is not a minimal cover, and so we must find one. We first replace $C \rightarrow CSJDPQV$ with the FDs:

$$C \rightarrow S, C \rightarrow J, C \rightarrow D, C \rightarrow P, C \rightarrow Q, \text{ and } C \rightarrow V.$$  

The FD $C \rightarrow P$ is implied by $C \rightarrow S$, $C \rightarrow D$, and $SD \rightarrow P$; so we can delete it. The FD $C \rightarrow S$ is implied by $C \rightarrow J$ and $J \rightarrow S$; so we can delete it. This leaves us with a minimal cover:

$$C \rightarrow J, C \rightarrow D, C \rightarrow Q, C \rightarrow V, JP \rightarrow C, SD \rightarrow P, \text{ and } J \rightarrow S.$$  

Using the algorithm for ensuring dependency-preservation, we obtain the relational schema $CJ$, $CD$, $CQ$, $CV$, $CJP$, $SDP$, and $JS$. We can improve this schema by combining relations for which $C$ is the key into $CDJPQV$. In addition, we have $SDP$ and $JS$ in our decomposition. Since one of these relations ($CDJPQV$) is a superkey, we are done.

Comparing this decomposition with the one that we obtained earlier in this section, we find that they are quite close, with the only difference being that one of them has $CDJPQV$ instead of $CJP$ and $CJDQV$. In general, however, there could be significant differences.

Database designers typically use a conceptual design methodology (e.g., ER design) to arrive at an initial database design. Given this, the approach of repeated decompositions to rectify instances of redundancy is likely to be the most natural use of FDs and normalization techniques. However, a designer can also consider the alternative designs suggested by the synthesis approach.

### 15.8 OTHER KINDS OF DEPENDENCIES *

FDs are probably the most common and important kind of constraint from the point of view of database design. However, there are several other kinds of dependencies. In particular, there is a well-developed theory for database design using *multivalued dependencies* and *join dependencies*. By taking such dependencies into account, we can identify potential redundancy problems that cannot be detected using FDs alone.

This section illustrates the kinds of redundancy that can be detected using multivalued dependencies. Our main observation, however, is that simple guidelines (which can be checked using only FD reasoning) can tell us whether we even need to worry about complex constraints such as multivalued and join dependencies. We also comment on the role of *inclusion dependencies* in database design.
15.8.1 Multivalued Dependencies

Suppose that we have a relation with attributes course, teacher, and book, which we denote as CTB. The meaning of a tuple is that teacher T can teach course C, and book B is a recommended text for the course. There are no FDs; the key is CTB. However, the recommended texts for a course are independent of the instructor. The instance shown in Figure 15.13 illustrates this situation.

<table>
<thead>
<tr>
<th>course</th>
<th>teacher</th>
<th>book</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physics101</td>
<td>Green</td>
<td>Mechanics</td>
</tr>
<tr>
<td>Physics101</td>
<td>Green</td>
<td>Optics</td>
</tr>
<tr>
<td>Physics101</td>
<td>Brown</td>
<td>Mechanics</td>
</tr>
<tr>
<td>Physics101</td>
<td>Brown</td>
<td>Optics</td>
</tr>
<tr>
<td>Math301</td>
<td>Green</td>
<td>Mechanics</td>
</tr>
<tr>
<td>Math301</td>
<td>Green</td>
<td>Vectors</td>
</tr>
<tr>
<td>Math301</td>
<td>Green</td>
<td>Geometry</td>
</tr>
</tbody>
</table>

Figure 15.13 BCNF Relation with Redundancy That Is Revealed by MVDs

There are three points to note here:

- The relation schema CTB is in BCNF; thus we would not consider decomposing it further if we looked only at the FDs that hold over CTB.
- There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.
- The redundancy can be eliminated by decomposing CTB into CT and CB.

The redundancy in this example is due to the constraint that the texts for a course are independent of the instructors, which cannot be expressed in terms of FDs. This constraint is an example of a multivalued dependency, or MVD. Ideally, we should model this situation using two binary relationship sets, Instructors with attributes CT and Text with attributes CB. Because these are two essentially independent relationships, modeling them with a single ternary relationship set with attributes CTB is inappropriate. (See Section 2.5.3 for a further discussion of ternary versus binary relationships.) Given the subjectivity of ER design, however, we might create a ternary relationship. A careful analysis of the MVD information would then reveal the problem.

Let R be a relation schema and let X and Y be subsets of the attributes of R. Intuitively, the multivalued dependency X \( \rightarrow \rightarrow \) Y is said to hold over R if, in every legal
instance \( r \) of \( R \), each \( X \) value is associated with a set of \( Y \) values and this set is independent of the values in the other attributes.

Formally, if the MVD \( X \rightarrow\rightarrow Y \) holds over \( R \) and \( Z = R - XY \), the following must be true for every legal instance \( r \) of \( R \):

\[
\text{If } t_1 \in r, t_2 \in r \text{ and } t_1.X = t_2.X, \text{ then there must be some } t_3 \in r \text{ such that } t_1.XY = t_3.XY \text{ and } t_2.Z = t_3.Z.
\]

Figure 15.14 illustrates this definition. If we are given the first two tuples and told that the MVD \( X \rightarrow\rightarrow Y \) holds over this relation, we can infer that the relation instance must also contain the third tuple. Indeed, by interchanging the roles of the first two tuples—treating the first tuple as \( t_2 \) and the second tuple as \( t_1 \)—we can deduce that the tuple \( t_4 \) must also be in the relation instance.

\[
\begin{array}{|c|c|c|}
\hline
X & Y & Z \\
\hline
a & b_1 & c_1 & \text{— tuple } t_1 \\
\hline
a & b_2 & c_2 & \text{— tuple } t_2 \\
\hline
a & b_1 & c_2 & \text{— tuple } t_3 \\
\hline
a & b_2 & c_1 & \text{— tuple } t_4 \\
\hline
\end{array}
\]

\text{Figure 15.14 Illustration of MVD Definition}

This table suggests another way to think about MVDs: If \( X \rightarrow\rightarrow Y \) holds over \( R \), then \( \pi_Y(\sigma_{X=x}(R)) = \pi_Y(\sigma_{X=x}(R)) \times \pi_Z(\sigma_{X=x}(R)) \) in every legal instance of \( R \), for any value \( x \) that appears in the \( X \) column of \( R \). In other words, consider groups of tuples in \( R \) with the same \( X \)-value, for each \( X \)-value. In each such group consider the projection onto the attributes \( YZ \). This projection must be equal to the cross-product of the projections onto \( Y \) and \( Z \). That is, for a given \( X \)-value, the \( Y \)-values and \( Z \)-values are independent. (From this definition it is easy to see that \( X \rightarrow\rightarrow Y \) must hold whenever \( X \rightarrow Y \) holds. If the FD \( X \rightarrow Y \) holds, there is exactly one \( Y \)-value for a given \( X \)-value, and the conditions in the MVD definition hold trivially. The converse does not hold, as Figure 15.14 illustrates.)

Returning to our \( CTB \) example, the constraint that course texts are independent of instructors can be expressed as \( C \rightarrow\rightarrow T \). In terms of the definition of MVDs, this constraint can be read as follows:

\text{If (there is a tuple showing that) } C \text{ is taught by teacher } T, \text{ and (there is a tuple showing that) } C \text{ has book } B \text{ as text, then (there is a tuple showing that) } C \text{ is taught by } T \text{ and has text } B.\)
Given a set of FDs and MVDs, in general we can infer that several additional FDs and MVDs hold. A sound and complete set of inference rules consists of the three Armstrong Axioms plus five additional rules. Three of the additional rules involve only MVDs:

- **MVD Complementation**: If $X \rightarrow\rightarrow Y$, then $X \rightarrow\rightarrow R - XY$.
- **MVD Augmentation**: If $X \rightarrow\rightarrow Y$ and $W \supseteq Z$, then $WX \rightarrow\rightarrow YZ$.
- **MVD Transitivity**: If $X \rightarrow\rightarrow Y$ and $Y \rightarrow\rightarrow Z$, then $X \rightarrow\rightarrow (Z - Y)$.

As an example of the use of these rules, since we have $C \rightarrow\rightarrow T$ over $CTB$, MVD complementation allows us to infer that $C \rightarrow\rightarrow CTB - CT$ as well, that is, $C \rightarrow\rightarrow B$. The remaining two rules relate FDs and MVDs:

- **Replication**: If $X \rightarrow Y$, then $X \rightarrow\rightarrow Y$.
- **Coalescence**: If $X \rightarrow\rightarrow Y$ and there is a $W$ such that $W \cap Y$ is empty, $W \rightarrow Z$, and $Y \supseteq Z$, then $X \rightarrow Z$.

Observe that replication states that every FD is also an MVD.

### 15.8.2 Fourth Normal Form

Fourth normal form is a direct generalization of BCNF. Let $R$ be a relation schema, $X$ and $Y$ be nonempty subsets of the attributes of $R$, and $F$ be a set of dependencies that includes both FDs and MVDs. $R$ is said to be in **fourth normal form (4NF)** if for every MVD $X \rightarrow\rightarrow Y$ that holds over $R$, one of the following statements is true:

- $Y \subseteq X$ or $XY = R$, or
- $X$ is a superkey.

In reading this definition, it is important to understand that the definition of a key has not changed—the key must uniquely determine all attributes through FDs alone. $X \rightarrow Y$ is a **trivial MVD** if $Y \subseteq X \subseteq R$ or $XY = R$; such MVDs always hold.

The relation $CTB$ is not in 4NF because $C \rightarrow\rightarrow T$ is a nontrivial MVD and $C$ is not a key. We can eliminate the resulting redundancy by decomposing $CTB$ into $CT$ and $CB$; each of these relations is then in 4NF.

To use MVD information fully, we must understand the theory of MVDs. However, the following result due to Date and Fagin identifies conditions—detected using only FD information!—under which we can safely ignore MVD information. That is, using MVD information in addition to the FD information will not reveal any redundancy. Therefore, if these conditions hold, we do not even need to identify all MVDs.
If a relation schema is in BCNF, and at least one of its keys consists of a single attribute, it is also in 4NF.

An important assumption is implicit in any application of the preceding result: *The set of FDs identified thus far is indeed the set of all FDs that hold over the relation.* This assumption is important because the result relies on the relation being in BCNF, which in turn depends on the set of FDs that hold over the relation.

We illustrate this point using an example. Consider a relation schema $ABCD$ and suppose that the FD $A \rightarrow BCD$ and the MVD $B \rightarrow\rightarrow C$ are given. Considering only these dependencies, this relation schema appears to be a counter example to the result. The relation has a simple key, appears to be in BCNF, and yet is not in 4NF because $B \rightarrow\rightarrow C$ causes a violation of the 4NF conditions. But let’s take a closer look.

Figure 15.15 shows three tuples from an instance of $ABCD$ that satisfies the given MVD $B \rightarrow\rightarrow C$. From the definition of an MVD, given tuples $t_1$ and $t_2$, it follows

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>$C$</td>
<td>$A$</td>
<td>$D$</td>
</tr>
<tr>
<td>$b$</td>
<td>$c_1$</td>
<td>$a_1$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$b$</td>
<td>$c_2$</td>
<td>$a_2$</td>
<td>$d_2$</td>
</tr>
<tr>
<td>$b$</td>
<td>$c_1$</td>
<td>$a_2$</td>
<td>$d_2$</td>
</tr>
</tbody>
</table>

*Figure 15.15  Three Tuples from a Legal Instance of $ABCD$*

that tuple $t_3$ must also be included in the instance. Consider tuples $t_2$ and $t_3$. From the given FD $A \rightarrow BCD$ and the fact that these tuples have the same $A$-value, we can deduce that $c_1 = c_2$. Thus, we see that the FD $B \rightarrow C$ must hold over $ABCD$ whenever the FD $A \rightarrow BCD$ and the MVD $B \rightarrow\rightarrow C$ hold. If $B \rightarrow C$ holds, the relation $ABCD$ is not in BCNF (unless additional FDs hold that make $B$ a key)!

Thus, the apparent counter example is really not a counter example—rather, it illustrates the importance of correctly identifying all FDs that hold over a relation. In this example $A \rightarrow BCD$ is not the only FD; the FD $B \rightarrow C$ also holds but was not identified initially. Given a set of FDs and MVDs, the inference rules can be used to infer additional FDs (and MVDs); to apply the Date-Fagin result without first using the MVD inference rules, we must be certain that we have identified all the FDs.

In summary, the Date-Fagin result offers a convenient way to check that a relation is in 4NF (without reasoning about MVDs) if we are confident that we have identified all FDs. At this point the reader is invited to go over the examples we have discussed in this chapter and see if there is a relation that is not in 4NF.
15.8.3 Join Dependencies

A join dependency is a further generalization of MVDs. A join dependency (JD) \( \preceq \{R_1, \ldots, R_n\} \) is said to hold over a relation \( R \) if \( R_1, \ldots, R_n \) is a lossless-join decomposition of \( R \).

An MVD \( X \rightarrow Y \) over a relation \( R \) can be expressed as the join dependency \( \preceq \{XY, X(R-Y)\} \). As an example, in the CTB relation, the MVD \( C \rightarrow T \) can be expressed as the join dependency \( \preceq \{CT, CB\} \).

Unlike FDs and MVDs, there is no set of sound and complete inference rules for JDs.

15.8.4 Fifth Normal Form

A relation schema \( R \) is said to be in fifth normal form (5NF) if for every JD \( \preceq \{R_1, \ldots, R_n\} \) that holds over \( R \), one of the following statements is true:

- \( R_i = R \) for some \( i \), or
- The JD is implied by the set of those FDs over \( R \) in which the left side is a key for \( R \).

The second condition deserves some explanation, since we have not presented inference rules for FDs and JDs taken together. Intuitively, we must be able to show that the decomposition of \( R \) into \( \{R_1, \ldots, R_n\} \) is lossless-join whenever the key dependencies (FDs in which the left side is a key for \( R \)) hold. \( \preceq \{R_1, \ldots, R_n\} \) is a trivial JD if \( R_i = R \) for some \( i \); such a JD always holds.

The following result, also due to Date and Fagin, identifies conditions—again, detected using only FD information—under which we can safely ignore JD information.

If a relation schema is in 3NF and each of its keys consists of a single attribute, it is also in 5NF.

The conditions identified in this result are sufficient for a relation to be in 5NF, but not necessary. The result can be very useful in practice because it allows us to conclude that a relation is in 5NF without ever identifying the MVDs and JDs that may hold over the relation.

15.8.5 Inclusion Dependencies

MVDs and JDs can be used to guide database design, as we have seen, although they are less common than FDs and harder to recognize and reason about. In contrast,
inclusion dependencies are very intuitive and quite common. However, they typically have little influence on database design (beyond the ER design stage).

Informally, an inclusion dependency is a statement of the form that some columns of a relation are contained in other columns (usually of a second relation). A foreign key constraint is an example of an inclusion dependency; the referring column(s) in one relation must be contained in the primary key column(s) of the referenced relation. As another example, if $R$ and $S$ are two relations obtained by translating two entity sets such that every $R$ entity is also an $S$ entity, we would have an inclusion dependency; projecting $R$ on its key attributes yields a relation that is contained in the relation obtained by projecting $S$ on its key attributes.

The main point to bear in mind is that we should not split groups of attributes that participate in an inclusion dependency. For example, if we have an inclusion dependency $AB \subseteq CD$, while decomposing the relation schema containing $AB$, we should ensure that at least one of the schemas obtained in the decomposition contains both $A$ and $B$. Otherwise, we cannot check the inclusion dependency $AB \subseteq CD$ without reconstructing the relation containing $AB$.

Most inclusion dependencies in practice are key-based, that is, involve only keys. Foreign key constraints are a good example of key-based inclusion dependencies. An ER diagram that involves ISA hierarchies also leads to key-based inclusion dependencies. If all inclusion dependencies are key-based, we rarely have to worry about splitting attribute groups that participate in inclusions, since decompositions usually do not split the primary key. Note, however, that going from 3NF to BCNF always involves splitting some key (hopefully not the primary key!), since the dependency guiding the split is of the form $X \rightarrow A$ where $A$ is part of a key.

15.9 POINTS TO REVIEW

- Redundancy, storing the same information several times in a database, can result in update anomalies (all copies need to be updated), insertion anomalies (certain information cannot be stored unless other information is stored as well), and deletion anomalies (deleting some information means loss of other information as well). We can reduce redundancy by replacing a relation schema $R$ with several smaller relation schemas. This process is called decomposition. (Section 15.1)

- A functional dependency $X \rightarrow Y$ is a type of IC. It says that if two tuples agree upon (i.e., have the same) values in the $X$ attributes, then they also agree upon the values in the $Y$ attributes. (Section 15.2)

- FDs can help to refine subjective decisions made during conceptual design. (Section 15.3)
An FD $f$ is implied by a set $F$ of FDs if for all relation instances where $F$ holds, $f$ also holds. The closure of a set $F$ of FDs is the set of all FDs $F^+$ implied by $F$. Armstrong’s Axioms are a sound and complete set of rules to generate all FDs in the closure. An FD $X \rightarrow Y$ is trivial if $X$ contains only attributes that also appear in $Y$. The attribute closure $X^+$ of a set of attributes $X$ with respect to a set of FDs $F$ is the set of attributes $A$ such that $X \rightarrow A$ can be inferred using Armstrong’s Axioms. (Section 15.4)

A normal form is a property of a relation schema indicating the type of redundancy that the relation schema exhibits. If a relation schema is in Boyce-Codd normal form (BCNF), then the only nontrivial FDs are key constraints. If a relation is in third normal form (3NF), then all nontrivial FDs are key constraints or their right side is part of a candidate key. Thus, every relation that is in BCNF is also in 3NF, but not vice versa. (Section 15.5)

A decomposition of a relation schema $R$ into two relation schemas $X$ and $Y$ is a lossless-join decomposition with respect to a set of FDs $F$ if for any instance $r$ of $R$ that satisfies the FDs in $F$, $\pi_X(r) \bowtie \pi_Y(r) = r$. The decomposition of $R$ into $X$ and $Y$ is lossless-join if and only if $F^+$ contains either $X \cap Y \rightarrow X$ or the FD $X \cap Y \rightarrow Y$. The decomposition is dependency-preserving if we can enforce all FDs that are given to hold on $R$ by simply enforcing FDs on $X$ and FDs on $Y$ independently (i.e., without joining $X$ and $Y$). (Section 15.6)

There is an algorithm to obtain a lossless-join decomposition of a relation into a collection of BCNF relation schemas, but sometimes there is no dependency-preserving decomposition into BCNF schemas. We also discussed an algorithm for decomposing a relation schema into a collection of 3NF relation schemas. There is always a lossless-join, dependency-preserving decomposition into a collection of 3NF relation schemas. A minimal cover of a set of FDs is an equivalent set of FDs that has certain minimality properties (intuitively, the set of FDs is as small as possible). Instead of decomposing a relation schema, we can also synthesize a corresponding collection of 3NF relation schemas. (Section 15.7)

Other kinds of dependencies include multivalued dependencies, join dependencies, and inclusion dependencies. Fourth and fifth normal forms are more stringent than BCNF, and eliminate redundancy due to multivalued and join dependencies, respectively. (Section 15.8)

**EXERCISES**

**Exercise 15.1** Briefly answer the following questions.

1. Define the term functional dependency.
2. Give a set of FDs for the relation schema $R(A,B,C,D)$ with primary key $AB$ under which $R$ is in 1NF but not in 2NF.
3. Give a set of FDs for the relation schema $R(A,B,C,D)$ with primary key $AB$ under which $R$ is in 2NF but not in 3NF.

4. Consider the relation schema $R(A,B,C)$, which has the FD $B \rightarrow C$. If $A$ is a candidate key for $R$, is it possible for $R$ to be in BCNF? If so, under what conditions? If not, explain why not.

5. Suppose that we have a relation schema $R(A,B,C)$ representing a relationship between two entity sets with keys $A$ and $B$, respectively, and suppose that $R$ has (among others) the FDs $A \rightarrow B$ and $B \rightarrow A$. Explain what such a pair of dependencies means (i.e., what they imply about the relationship that the relation models).

**Exercise 15.2** Consider a relation $R$ with five attributes $ABCDE$. You are given the following dependencies: $A \rightarrow B$, $BC \rightarrow E$, and $ED \rightarrow A$.

1. List all keys for $R$.
2. Is $R$ in 3NF?
3. Is $R$ in BCNF?

**Exercise 15.3** Consider the following collection of relations and dependencies. Assume that each relation is obtained through decomposition from a relation with attributes $ABCDEFGHI$ and that all the known dependencies over relation $ABCDEFGHI$ are listed for each question. (The questions are independent of each other, obviously, since the given dependencies over $ABCDEFGHI$ are different.) For each (sub) relation: (a) State the strongest normal form that the relation is in. (b) If it is not in BCNF, decompose it into a collection of BCNF relations.

1. $R1(A,C,B,D,E)$, $A \rightarrow B$, $C \rightarrow D$
2. $R2(A,B,F)$, $AC \rightarrow E$, $B \rightarrow F$
3. $R3(A,D)$, $D \rightarrow G$, $G \rightarrow H$
4. $R4(D,C,H,G)$, $A \rightarrow I$, $I \rightarrow A$
5. $R5(A,I,C,E)$

**Exercise 15.4** Suppose that we have the following three tuples in a legal instance of a relation schema $S$ with three attributes $ABC$ (listed in order): $(1,2,3)$, $(4,2,3)$, and $(5,3,3)$.

1. Which of the following dependencies can you infer does not hold over schema $S$?
   - (a) $A \rightarrow B$
   - (b) $BC \rightarrow A$
   - (c) $B \rightarrow C$

2. Can you identify any dependencies that hold over $S$?

**Exercise 15.5** Suppose you are given a relation $R$ with four attributes, $ABCD$. For each of the following sets of FDs, assuming those are the only dependencies that hold for $R$, do the following: (a) Identify the candidate key(s) for $R$. (b) Identify the best normal form that $R$ satisfies (1NF, 2NF, 3NF, or BCNF). (c) If $R$ is not in BCNF, decompose it into a set of BCNF relations that preserve the dependencies.

1. $C \rightarrow D$, $C \rightarrow A$, $B \rightarrow C$
2. $B \rightarrow C, D \rightarrow A$
3. $ABC \rightarrow D, D \rightarrow A$
4. $A \rightarrow B, BC \rightarrow D, A \rightarrow C$
5. $AB \rightarrow C, AB \rightarrow D, C \rightarrow A, D \rightarrow B$

**Exercise 15.6** Consider the attribute set $R = ABCDEGH$ and the FD set $F = \{AB \rightarrow C, AC \rightarrow B, AD \rightarrow E, B \rightarrow D, BC \rightarrow A, E \rightarrow G\}$.

1. For each of the following attribute sets, do the following: (i) Compute the set of dependencies that hold over the set and write down a minimal cover. (ii) Name the strongest normal form that is not violated by the relation containing these attributes. (iii) Decompose it into a collection of BCNF relations if it is not in BCNF.
   (a) $ABC$
   (b) $ABCD$
   (c) $ABCEG$
   (d) $DCEGH$
   (e) $ACEH$

2. Which of the following decompositions of $R = ABCDEG$, with the same set of dependencies $F$, is (a) dependency-preserving? (b) lossless-join?
   (a) $\{AB, BC, ABDE, EG\}$
   (b) $\{ABC, ACDE, ADG\}$

**Exercise 15.7** Let $R$ be decomposed into $R_1, R_2, \ldots, R_n$. Let $F$ be a set of FDs on $R$.

1. Define what it means for $F$ to be preserved in the set of decomposed relations.
2. Describe a polynomial-time algorithm to test dependency-preservation.
3. Projecting the FDs stated over a set of attributes $X$ onto a subset of attributes $Y$ requires that we consider the closure of the FDs. Give an example where considering the closure is important in testing dependency-preservation; that is, considering just the given FDs gives incorrect results.

**Exercise 15.8** Consider a relation $R$ that has three attributes $ABC$. It is decomposed into relations $R_1$ with attributes $AB$ and $R_2$ with attributes $BC$.

1. State the definition of a lossless-join decomposition with respect to this example. Answer this question concisely by writing a relational algebra equation involving $R, R_1$, and $R_2$.
2. Suppose that $B \rightarrow C$. Is the decomposition of $R$ into $R_1$ and $R_2$ lossless-join? Reconcile your answer with the observation that neither of the FDs $R_1 \cap R_2 \rightarrow R_1$ nor $R_1 \cap R_2 \rightarrow R_2$ hold, in light of the simple test offering a necessary and sufficient condition for lossless-join decomposition into two relations in Section 15.6.1.
3. If you are given the following instances of $R_1$ and $R_2$, what can you say about the instance of $R$ from which these were obtained? Answer this question by listing tuples that are definitely in $R$ and listing tuples that are possibly in $R$.
   Instance of $R_1 = \{(5,1), (6,1)\}$
   Instance of $R_2 = \{(1,8), (1,9)\}$
   Can you say that attribute $B$ definitely is or is not a key for $R$?
Exercise 15.9 Suppose you are given a relation \( R(A,B,C,D) \). For each of the following sets of FDs, assuming they are the only dependencies that hold for \( R \), do the following: (a) Identify the candidate key(s) for \( R \). (b) State whether or not the proposed decomposition of \( R \) into smaller relations is a good decomposition, and briefly explain why or why not.

1. \( B \rightarrow C, D \rightarrow A \); decompose into \( BC \) and \( AD \).
2. \( AB \rightarrow C, C \rightarrow A, C \rightarrow D \); decompose into \( ACD \) and \( BC \).
3. \( A \rightarrow BC, C \rightarrow AD \); decompose into \( ABC \) and \( AD \).
4. \( A \rightarrow B, B \rightarrow C, C \rightarrow D \); decompose into \( AB \) and \( ACD \).
5. \( A \rightarrow B, B \rightarrow C, C \rightarrow D \); decompose into \( AB, AD \) and \( CD \).

Exercise 15.10 Suppose that we have the following four tuples in a relation \( S \) with three attributes \( ABC \): \((1,2,3), (4,2,3), (5,3,3), (5,3,4)\). Which of the following functional (\( \rightarrow \)) and multivalued (\( \rightarrow \rightarrow \)) dependencies can you infer does not hold over relation \( S \)?

1. \( A \rightarrow B \)
2. \( A \rightarrow \rightarrow B \)
3. \( BC \rightarrow A \)
4. \( BC \rightarrow \rightarrow A \)
5. \( B \rightarrow C \)
6. \( B \rightarrow \rightarrow C \)

Exercise 15.11 Consider a relation \( R \) with five attributes \( ABCDE \).

1. For each of the following instances of \( R \), state whether (a) it violates the FD \( BC \rightarrow D \), and (b) it violates the MVD \( BC \rightarrow \rightarrow D \):
   - (a) \( \{ \} \) (i.e., empty relation)
   - (b) \( \{ (a,2,3,4,5), (2,a,3,5,5) \} \)
   - (c) \( \{ (a,2,3,4,5), (2,a,3,5,5), (a,2,3,4,6) \} \)
   - (d) \( \{ (a,2,3,4,5), (2,a,3,4,5), (a,2,3,6,5) \} \)
   - (e) \( \{ (a,2,3,4,5), (2,a,3,7,5), (a,2,3,4,6) \} \)
   - (f) \( \{ (a,2,3,4,5), (2,a,3,4,5), (a,2,3,6,5), (a,2,3,6,6) \} \)
   - (g) \( \{ (a,2,3,4,5), (a,2,3,6,5), (a,2,3,6,6), (a,2,3,4,6) \} \)

2. If each instance for \( R \) listed above is legal, what can you say about the FD \( A \rightarrow B \)?

Exercise 15.12 JDs are motivated by the fact that sometimes a relation that cannot be decomposed into two smaller relations in a lossless-join manner can be so decomposed into three or more relations. An example is a relation with attributes \textit{supplier, part, and project}, denoted \( SPJ \), with no FDs or MVDs. The JD \( \Join \{ SP, PJ, JS \} \) holds.

From the JD, the set of relation schemes \( SP, PJ, \) and \( JS \) is a lossless-join decomposition of \( SPJ \). Construct an instance of \( SPJ \) to illustrate that no two of these schemes suffice.
Exercise 15.13 Consider a relation $R$ with attributes $ABCDE$. Let the following FDs be given: $A \rightarrow BC$, $BC \rightarrow E$, and $E \rightarrow DA$. Similarly, let $S$ be a relation with attributes $ABCDE$ and let the following FDs be given: $A \rightarrow BC$, $B \rightarrow E$, and $E \rightarrow DA$. (Only the second dependency differs from those that hold over $R$.) You do not know whether or which other (join) dependencies hold.

1. Is $R$ in BCNF?
2. Is $R$ in 4NF?
3. Is $R$ in 5NF?
4. Is $S$ in BCNF?
5. Is $S$ in 4NF?
6. Is $S$ in 5NF?

Exercise 15.14 Let us say that an FD $X \rightarrow Y$ is simple if $Y$ is a single attribute.

1. Replace the FD $AB \rightarrow CD$ by the smallest equivalent collection of simple FDs.
2. Prove that every FD $X \rightarrow Y$ in a set of FDs $F$ can be replaced by a set of simple FDs such that $F^+$ is equal to the closure of the new set of FDs.

Exercise 15.15 Prove that Armstrong’s Axioms are sound and complete for FD inference. That is, show that repeated application of these axioms on a set $F$ of FDs produces exactly the dependencies in $F^+$.

Exercise 15.16 Describe a linear-time (in the size of the set of FDs, where the size of each FD is the number of attributes involved) algorithm for finding the attribute closure of a set of attributes with respect to a set of FDs.

Exercise 15.17 Consider a scheme $R$ with FDs $F$ that is decomposed into schemes with attributes $X$ and $Y$. Show that this is dependency-preserving if $F \subseteq (F_X \cup F_Y)^+$.

Exercise 15.18 Let $R$ be a relation schema with a set $F$ of FDs. Prove that the decomposition of $R$ into $R_1$ and $R_2$ is lossless-join if and only if $F^+$ contains $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$.

Exercise 15.19 Prove that the optimization of the algorithm for lossless-join, dependency-preserving decomposition into 3NF relations (Section 15.7.2) is correct.

Exercise 15.20 Prove that the 3NF synthesis algorithm produces a lossless-join decomposition of the relation containing all the original attributes.

Exercise 15.21 Prove that an MVD $X \rightarrow\rightarrow Y$ over a relation $R$ can be expressed as the join dependency $\bowtie \{XY, X(R - Y)\}$.

Exercise 15.22 Prove that if $R$ has only one key, it is in BCNF if and only if it is in 3NF.

Exercise 15.23 Prove that if $R$ is in 3NF and every key is simple, then $R$ is in BCNF.

Exercise 15.24 Prove these statements:

1. If a relation scheme is in BCNF and at least one of its keys consists of a single attribute, it is also in 4NF.
2. If a relation scheme is in 3NF and each key has a single attribute, it is also in 5NF.

Exercise 15.25 Give an algorithm for testing whether a relation scheme is in BCNF. The algorithm should be polynomial in the size of the set of given FDs. (The size is the sum over all FDs of the number of attributes that appear in the FD.) Is there a polynomial algorithm for testing whether a relation scheme is in 3NF?
Chapter 15

PROJECT-BASED EXERCISES

Exercise 15.26 Minibase provides a tool called Designview for doing database design using FDs. It lets you check whether a relation is in a particular normal form, test whether decompositions have nice properties, compute attribute closures, try several decomposition sequences and switch between them, generate SQL statements to create the final database schema, and so on.

1. Use Designview to check your answers to exercises that call for computing closures, testing normal forms, decomposing into a desired normal form, and so on.

2. (Note to instructors: This exercise should be made more specific by providing additional details. See Appendix B.) Apply Designview to a large, real-world design problem.

BIBLIOGRAPHIC NOTES

Textbook presentations of dependency theory and its use in database design include [3, 38, 436, 443, 656]. Good survey articles on the topic include [663, 355].

FDs were introduced in [156], along with the concept of 3NF, and axioms for inferring FDs were presented in [31]. BCNF was introduced in [157]. The concept of a legal relation instance and dependency satisfaction are studied formally in [279]. FDs were generalized to semantic data models in [674].

Finding a key is shown to be NP-complete in [432]. Lossless-join decompositions were studied in [24, 437, 546]. Dependency-preserving decompositions were studied in [61]. [68] introduced minimal covers. Decomposition into 3NF is studied by [68, 85] and decomposition into BCNF is addressed in [651]. [351] shows that testing whether a relation is in 3NF is NP-complete. [215] introduced 4NF and discussed decomposition into 4NF. Fagin introduced other normal forms in [216] (project-join normal form) and [217] (domain-key normal form). In contrast to the extensive study of vertical decompositions, there has been relatively little formal investigation of horizontal decompositions. [175] investigates horizontal decompositions.

MVDs were discovered independently by Delobel [177], Fagin [215], and Zaniolo [690]. Axioms for FDs and MVDs were presented in [60]. [516] shows that there is no axiomatization for JDs, although [575] provides an axiomatization for a more general class of dependencies. The sufficient conditions for 4NF and 5NF in terms of FDs that were discussed in Section 15.8 are from [171]. An approach to database design that uses dependency information to construct sample relation instances is described in [442, 443].
Advice to a client who complained about rain leaking through the roof onto the dining table: “Move the table.”

—Architect Frank Lloyd Wright

The performance of a DBMS on commonly asked queries and typical update operations is the ultimate measure of a database design. A DBA can improve performance by adjusting some DBMS parameters (e.g., the size of the buffer pool or the frequency of checkpointing) and by identifying performance bottlenecks and adding hardware to eliminate such bottlenecks. The first step in achieving good performance, however, is to make good database design choices, which is the focus of this chapter.

After we have designed the conceptual and external schemas, that is, created a collection of relations and views along with a set of integrity constraints, we must address performance goals through physical database design, in which we design the physical schema. As user requirements evolve, it is usually necessary to tune, or adjust, all aspects of a database design for good performance.

This chapter is organized as follows. We give an overview of physical database design and tuning in Section 16.1. The most important physical design decisions concern the choice of indexes. We present guidelines for deciding which indexes to create in Section 16.2. These guidelines are illustrated through several examples and developed further in Sections 16.3 through 16.6. In Section 16.3 we present examples that highlight basic alternatives in index selection. In Section 16.4 we look closely at the important issue of clustering; we discuss how to choose clustered indexes and whether to store tuples from different relations near each other (an option supported by some DBMSs). In Section 16.5 we consider the use of indexes with composite or multiple-attribute search keys. In Section 16.6 we emphasize how well-chosen indexes can enable some queries to be answered without ever looking at the actual data records.

In Section 16.7 we survey the main issues of database tuning. In addition to tuning indexes, we may have to tune the conceptual schema, as well as frequently used query and view definitions. We discuss how to refine the conceptual schema in Section 16.8 and how to refine queries and view definitions in Section 16.9. We briefly discuss the performance impact of concurrent access in Section 16.10. We conclude the chap-
ter with a short discussion of DBMS benchmarks in Section 16.11; benchmarks help evaluate the performance of alternative DBMS products.

16.1 INTRODUCTION TO PHYSICAL DATABASE DESIGN

Like all other aspects of database design, physical design must be guided by the nature of the data and its intended use. In particular, it is important to understand the typical workload that the database must support; the workload consists of a mix of queries and updates. Users also have certain requirements about how fast certain queries or updates must run or how many transactions must be processed per second. The workload description and users’ performance requirements are the basis on which a number of decisions have to be made during physical database design.

To create a good physical database design and to tune the system for performance in response to evolving user requirements, the designer needs to understand the workings of a DBMS, especially the indexing and query processing techniques supported by the DBMS. If the database is expected to be accessed concurrently by many users, or is a distributed database, the task becomes more complicated, and other features of a DBMS come into play. We discuss the impact of concurrency on database design in Section 16.10. We discuss distributed databases in Chapter 21.

16.1.1 Database Workloads

The key to good physical design is arriving at an accurate description of the expected workload. A workload description includes the following elements:

1. A list of queries and their frequencies, as a fraction of all queries and updates.
2. A list of updates and their frequencies.
3. Performance goals for each type of query and update.

For each query in the workload, we must identify:
Physical Database Design and Tuning

- Which relations are accessed.
- Which attributes are retained (in the `SELECT` clause).
- Which attributes have selection or join conditions expressed on them (in the `WHERE` clause) and how selective these conditions are likely to be.

Similarly, for each update in the workload, we must identify:

- Which attributes have selection or join conditions expressed on them (in the `WHERE` clause) and how selective these conditions are likely to be.
- The type of update (`INSERT`, `DELETE`, or `UPDATE`) and the updated relation.
- For `UPDATE` commands, the fields that are modified by the update.

Remember that queries and updates typically have parameters, for example, a debit or credit operation involves a particular account number. The values of these parameters determine selectivity of selection and join conditions.

Updates have a query component that is used to find the target tuples. This component can benefit from a good physical design and the presence of indexes. On the other hand, updates typically require additional work to maintain indexes on the attributes that they modify. Thus, while queries can only benefit from the presence of an index, an index may either speed up or slow down a given update. Designers should keep this trade-off in mind when creating indexes.

16.1.2 Physical Design and Tuning Decisions

Important decisions made during physical database design and database tuning include the following:

1. *Which indexes to create.*
   - Which relations to index and which field or combination of fields to choose as index search keys.
   - For each index, should it be clustered or unclustered? Should it be dense or sparse?

2. *Whether we should make changes to the conceptual schema in order to enhance performance.* For example, we have to consider:
   - *Alternative normalized schemas:* We usually have more than one way to decompose a schema into a desired normal form (BCNF or 3NF). A choice can be made on the basis of performance criteria.
Denormalization: We might want to reconsider schema decompositions carried out for normalization during the conceptual schema design process to improve the performance of queries that involve attributes from several previously decomposed relations.

Vertical partitioning: Under certain circumstances we might want to further decompose relations to improve the performance of queries that involve only a few attributes.

Views: We might want to add some views to mask the changes in the conceptual schema from users.

3. Whether frequently executed queries and transactions should be rewritten to run faster.

In parallel or distributed databases, which we discuss in Chapter 21, there are additional choices to consider, such as whether to partition a relation across different sites or whether to store copies of a relation at multiple sites.

16.1.3 Need for Database Tuning

Accurate, detailed workload information may be hard to come by while doing the initial design of the system. Consequently, tuning a database after it has been designed and deployed is important—we must refine the initial design in the light of actual usage patterns to obtain the best possible performance.

The distinction between database design and database tuning is somewhat arbitrary. We could consider the design process to be over once an initial conceptual schema is designed and a set of indexing and clustering decisions is made. Any subsequent changes to the conceptual schema or the indexes, say, would then be regarded as a tuning activity. Alternatively, we could consider some refinement of the conceptual schema (and physical design decisions affected by this refinement) to be part of the physical design process.

Where we draw the line between design and tuning is not very important, and we will simply discuss the issues of index selection and database tuning without regard to when the tuning activities are carried out.

16.2 GUIDELINES FOR INDEX SELECTION

In considering which indexes to create, we begin with the list of queries (including queries that appear as part of update operations). Obviously, only relations accessed by some query need to be considered as candidates for indexing, and the choice of attributes to index on is guided by the conditions that appear in the WHERE clauses of
the queries in the workload. The presence of suitable indexes can significantly improve
the evaluation plan for a query, as we saw in Chapter 13.

One approach to index selection is to consider the most important queries in turn, and
for each to determine which plan the optimizer would choose given the indexes that
are currently on our list of (to be created) indexes. Then we consider whether we can
arrive at a substantially better plan by adding more indexes; if so, these additional
indexes are candidates for inclusion in our list of indexes. In general, range retrievals
will benefit from a B+ tree index, and exact-match retrievals will benefit from a hash
index. Clustering will benefit range queries, and it will benefit exact-match queries if
several data entries contain the same key value.

Before adding an index to the list, however, we must consider the impact of having
this index on the updates in our workload. As we noted earlier, although an index can
speed up the query component of an update, all indexes on an updated attribute—on
any attribute, in the case of inserts and deletes—must be updated whenever the value
of the attribute is changed. Therefore, we must sometimes consider the trade-off of
slowing some update operations in the workload in order to speed up some queries.

Clearly, choosing a good set of indexes for a given workload requires an understanding
of the available indexing techniques, and of the workings of the query optimizer. The
following guidelines for index selection summarize our discussion:

**Guideline 1 (whether to index):** The obvious points are often the most important.
Don’t build an index unless some query—including the query components of updates—
will benefit from it. Whenever possible, choose indexes that speed up more than one
query.

**Guideline 2 (choice of search key):** Attributes mentioned in a \textit{WHERE} clause are
candidates for indexing.

- An exact-match selection condition suggests that we should consider an index on
the selected attributes, ideally, a hash index.

- A range selection condition suggests that we should consider a B+ tree (or ISAM)
index on the selected attributes. A B+ tree index is usually preferable to an ISAM
index. An ISAM index may be worth considering if the relation is infrequently
updated, but we will assume that a B+ tree index is always chosen over an ISAM
index, for simplicity.

**Guideline 3 (multiple-attribute search keys):** Indexes with multiple-attribute
search keys should be considered in the following two situations:

- A \textit{WHERE} clause includes conditions on more than one attribute of a relation.
They enable index-only evaluation strategies (i.e., accessing the relation can be avoided) for important queries. (This situation could lead to attributes being in the search key even if they do not appear in `WHERE` clauses.)

When creating indexes on search keys with multiple attributes, if range queries are expected, be careful to order the attributes in the search key to match the queries.

**Guideline 4 (whether to cluster):** At most one index on a given relation can be clustered, and clustering affects performance greatly; so the choice of clustered index is important.

- As a rule of thumb, range queries are likely to benefit the most from clustering. If several range queries are posed on a relation, involving different sets of attributes, consider the selectivity of the queries and their relative frequency in the workload when deciding which index should be clustered.
- If an index enables an index-only evaluation strategy for the query it is intended to speed up, the index need not be clustered. (Clustering matters only when the index is used to retrieve tuples from the underlying relation.)

**Guideline 5 (hash versus tree index):** A B+ tree index is usually preferable because it supports range queries as well as equality queries. A hash index is better in the following situations:

- The index is intended to support index nested loops join; the indexed relation is the inner relation, and the search key includes the join columns. In this case, the slight improvement of a hash index over a B+ tree for equality selections is magnified, because an equality selection is generated for each tuple in the outer relation.
- There is a very important equality query, and there are no range queries, involving the search key attributes.

**Guideline 6 (balancing the cost of index maintenance):** After drawing up a ‘wishlist’ of indexes to create, consider the impact of each index on the updates in the workload.

- If maintaining an index slows down frequent update operations, consider dropping the index.
- Keep in mind, however, that adding an index may well speed up a given update operation. For example, an index on employee ids could speed up the operation of increasing the salary of a given employee (specified by id).
16.3 BASIC EXAMPLES OF INDEX SELECTION

The following examples illustrate how to choose indexes during database design. The schemas used in the examples are not described in detail; in general they contain the attributes named in the queries. Additional information is presented when necessary.

Let us begin with a simple query:

```
SELECT E.ename, D.mgr
FROM Employees E, Departments D
WHERE D.dname='Toy' AND E.dno=D.dno
```

The relations mentioned in the query are Employees and Departments, and both conditions in the `WHERE` clause involve equalities. Our guidelines suggest that we should build hash indexes on the attributes involved. It seems clear that we should build a hash index on the `dname` attribute of Departments. But consider the equality `E.dno=D.dno`. Should we build an index (hash, of course) on the `dno` attribute of Departments or of Employees (or both)? Intuitively, we want to retrieve Departments tuples using the index on `dname` because few tuples are likely to satisfy the equality selection `D.dname='Toy'`. For each qualifying Departments tuple, we then find matching Employees tuples by using an index on the `dno` attribute of Employees. Thus, we should build an index on the `dno` field of Employees. (Note that nothing is gained by building an additional index on the `dno` field of Departments because Departments tuples are retrieved using the `dname` index.)

Our choice of indexes was guided by a query evaluation plan that we wanted to utilize. This consideration of a potential evaluation plan is common while making physical design decisions. Understanding query optimization is very useful for physical design. We show the desired plan for this query in Figure 16.1.

As a variant of this query, suppose that the `WHERE` clause is modified to be `WHERE D.dname='Toy' AND E.dno=D.dno AND E.age=25`. Let us consider alternative evaluation plans. One good plan is to retrieve Departments tuples that satisfy the selection on `dname` and to retrieve matching Employees tuples by using an index on the `dno` field; the selection on `age` is then applied on-the-fly. However, unlike the previous variant of this query, we do not really need to have an index on the `dno` field of Employees if we have an index on `age`. In this case we can retrieve Departments tuples that satisfy the selection on `dname` (by using the index on `dname`, as before), retrieve Employees tuples that satisfy the selection on `age` by using the index on `age`, and join these sets of tuples. Since the sets of tuples we join are small, they fit in memory and the join method is not important. This plan is likely to be somewhat poorer than using an

---

1This is only a heuristic. If `dname` is not the key, and we do not have statistics to verify this claim, it is possible that several tuples satisfy this condition!
index on \textit{dno}, but it is a reasonable alternative. Therefore, if we have an index on \textit{age} already (prompted by some other query in the workload), this variant of the sample query does not justify creating an index on the \textit{dno} field of \textit{Employees}.

Our next query involves a range selection:

\begin{verbatim}
SELECT E.ename, D.dname
FROM Employees E, Departments D
WHERE E.sal BETWEEN 10000 AND 20000
AND E.hobby='Stamps' AND E.dno=D.dno
\end{verbatim}

This query illustrates the use of the \texttt{BETWEEN} operator for expressing range selections. It is equivalent to the condition:

\begin{equation}
10000 \leq E\text{.sal} \text{ AND } E\text{.sal} \leq 20000
\end{equation}

The use of \texttt{BETWEEN} to express range conditions is recommended; it makes it easier for both the user and the optimizer to recognize both parts of the range selection.

Returning to the example query, both (nonjoin) selections are on the \textit{Employees} relation. Therefore, it is clear that a plan in which \textit{Employees} is the outer relation and \textit{Departments} is the inner relation is the best, as in the previous query, and we should build a hash index on the \textit{dno} attribute of \textit{Departments}. But which index should we build on \textit{Employees}? A B+ tree index on the \textit{sal} attribute would help with the range selection, especially if it is clustered. A hash index on the \textit{hobby} attribute would help with the equality selection. If one of these indexes is available, we could retrieve \textit{Employees} tuples using this index, retrieve matching \textit{Departments} tuples using the index on \textit{dno}, and apply all remaining selections and projections on-the-fly. If both indexes are available, the optimizer would choose the more selective access path for the given query; that is, it would consider which selection (the range condition on \textit{salary} or the equality on \textit{hobby}) has fewer qualifying tuples. In general, which access path is more
selective depends on the data. If there are very few people with salaries in the given range and many people collect stamps, the B+ tree index is best. Otherwise, the hash index on hobby is best.

If the query constants are known (as in our example), the selectivities can be estimated if statistics on the data are available. Otherwise, as a rule of thumb, an equality selection is likely to be more selective, and a reasonable decision would be to create a hash index on hobby. Sometimes, the query constants are not known—we might obtain a query by expanding a query on a view at run-time, or we might have a query in dynamic SQL, which allows constants to be specified as wild-card variables (e.g., %X) and instantiated at run-time (see Sections 5.9 and 5.10). In this case, if the query is very important, we might choose to create a B+ tree index on sal and a hash index on hobby and leave the choice to be made by the optimizer at run-time.

16.4 CLUSTERING AND INDEXING *

Range queries are good candidates for improvement with a clustered index:

\[
\text{SELECT E.dno FROM Employees E WHERE E.age > 40}
\]

If we have a B+ tree index on age, we can use it to retrieve only tuples that satisfy the selection \(E.age > 40\). Whether such an index is worthwhile depends first of all on the selectivity of the condition. What fraction of the employees are older than 40? If virtually everyone is older than 40, we don’t gain much by using an index on age; a sequential scan of the relation would do almost as well. However, suppose that only 10 percent of the employees are older than 40. Now, is an index useful? The answer depends on whether the index is clustered. If the index is unclustered, we could have one page I/O per qualifying employee, and this could be more expensive than a sequential scan even if only 10 percent of the employees qualify! On the other hand, a clustered B+ tree index on age requires only 10 percent of the I/Os for a sequential scan (ignoring the few I/Os needed to traverse from the root to the first retrieved leaf page and the I/Os for the relevant index leaf pages).

As another example, consider the following refinement of the previous query:

\[
\text{SELECT E.dno, COUNT(*) FROM Employees E WHERE E.age > 10 GROUP BY E.dno}
\]

If a B+ tree index is available on age, we could retrieve tuples using it, sort the retrieved tuples on dno, and so answer the query. However, this may not be a good
plan if virtually all employees are more than 10 years old. This plan is especially bad if the index is not clustered.

Let us consider whether an index on \( dno \) might suit our purposes better. We could use the index to retrieve all tuples, grouped by \( dno \), and for each \( dno \) count the number of tuples with \( age > 10 \). (This strategy can be used with both hash and B+ tree indexes; we only require the tuples to be \textit{grouped}, not necessarily \textit{sorted}, by \( dno \).) Again, the efficiency depends crucially on whether the index is clustered. If it is, this plan is likely to be the best if the condition on \( age \) is not very selective. (Even if we have a clustered index on \( age \), if the condition on \( age \) is not selective, the cost of sorting qualifying tuples on \( dno \) is likely to be high.) If the index is not clustered, we could perform one page I/O per tuple in Employees, and this plan would be terrible. Indeed, if the index is not clustered, the optimizer will choose the straightforward plan based on sorting on \( dno \). Thus, this query suggests that we build a clustered index on \( dno \) if the condition on \( age \) is not very selective. If the condition is very selective, we should consider building an index (not necessarily clustered) on \( age \) instead.

Clustering is also important for an index on a search key that does not include a candidate key, that is, an index in which several data entries can have the same key value. To illustrate this point, we present the following query:

\[
\text{SELECT E.dno} \\
\text{FROM Employees E} \\
\text{WHERE E.hobby='Stamps'}
\]

If many people collect stamps, retrieving tuples through an unclustered index on \( hobby \) can be very inefficient. It may be cheaper to simply scan the relation to retrieve all tuples and to apply the selection on-the-fly to the retrieved tuples. Therefore, if such a query is important, we should consider making the index on \( hobby \) a clustered index. On the other hand, if we assume that \( eid \) is a key for Employees, and replace the condition \( E.hobby='Stamps' \) by \( E.eid=552 \), we know that at most one Employees tuple will satisfy this selection condition. In this case, there is no advantage to making the index clustered.

Clustered indexes can be especially important while accessing the inner relation in an index nested loops join. To understand the relationship between clustered indexes and joins, let us revisit our first example.

\[
\text{SELECT E.ename, D.mgr} \\
\text{FROM Employees E, Departments D} \\
\text{WHERE D.dname='Toy' AND E.dno=D.dno}
\]

We concluded that a good evaluation plan is to use an index on \( dname \) to retrieve Departments tuples satisfying the condition on \( dname \) and to find matching Employees
tuples using an index on \textit{dno}. Should these indexes be clustered? Given our assumption that the number of tuples satisfying \( D.dname = 'Toy' \) is likely to be small, we should build an unclustered index on \textit{dname}. On the other hand, Employees is the inner relation in an index nested loops join, and \textit{dno} is not a candidate key. This situation is a strong argument that the index on the \textit{dno} field of Employees should be clustered. In fact, because the join consists of repeatedly posing equality selections on the \textit{dno} field of the inner relation, this type of query is a stronger justification for making the index on \textit{dno} be clustered than a simple selection query such as the previous selection on \textit{hobby}. (Of course, factors such as selectivities and frequency of queries have to be taken into account as well.)

The following example, very similar to the previous one, illustrates how clustered indexes can be used for sort-merge joins.

\begin{verbatim}
SELECT E.ename, D.mgr
FROM Employees E, Departments D
WHERE E.hobby='Stamps' AND E.dno=D.dno
\end{verbatim}

This query differs from the previous query in that the condition \( E.hobby='Stamps' \) replaces \( D.dname='Toy' \). Based on the assumption that there are few employees in the Toy department, we chose indexes that would facilitate an indexed nested loops join with Departments as the outer relation. Now let us suppose that many employees collect stamps. In this case, a block nested loops or sort-merge join might be more efficient. A sort-merge join can take advantage of a clustered B+ tree index on the \textit{dno} attribute in Departments to retrieve tuples and thereby avoid sorting Departments. Note that an unclustered index is not useful—since all tuples are retrieved, performing one I/O per tuple is likely to be prohibitively expensive. If there is no index on the \textit{dno} field of Employees, we could retrieve Employees tuples (possibly using an index on \textit{hobby}, especially if the index is clustered), apply the selection \( E.hobby='Stamps' \) on-the-fly, and sort the qualifying tuples on \textit{dno}.

As our discussion has indicated, when we retrieve tuples using an index, the impact of clustering depends on the number of retrieved tuples, that is, the number of tuples that satisfy the selection conditions that match the index. An unclustered index is just as good as a clustered index for a selection that retrieves a single tuple (e.g., an equality selection on a candidate key). As the number of retrieved tuples increases, the unclustered index quickly becomes more expensive than even a sequential scan of the entire relation. Although the sequential scan retrieves all tuples, it has the property that each page is retrieved exactly once, whereas a page may be retrieved as often as the number of tuples it contains if an unclustered index is used. If blocked I/O is performed (as is common), the relative advantage of sequential scan versus an unclustered index increases further. (Blocked I/O also speeds up access using a clustered index, of course.)
We illustrate the relationship between the number of retrieved tuples, viewed as a percentage of the total number of tuples in the relation, and the cost of various access methods in Figure 16.2. We assume that the query is a selection on a single relation, for simplicity. (Note that this figure reflects the cost of writing out the result; otherwise, the line for sequential scan would be flat.)

![Figure 16.2 The Impact of Clustering](image)

### 16.4.1 Co-clustering Two Relations

In our description of a typical database system architecture in Chapter 7, we explained how a relation is stored as a file of records. Although a file usually contains only the records of some one relation, some systems allow records from more than one relation to be stored in a single file. The database user can request that the records from two relations be interleaved physically in this manner. This data layout is sometimes referred to as **co-clustering** the two relations. We now discuss when co-clustering can be beneficial.

As an example, consider two relations with the following schemas:

- **Parts**
  - `pid: integer`
  - `pname: string`
  - `cost: integer`
  - `supplierid: integer`

- **Assembly**
  - `partid: integer`
  - `componentid: integer`
  - `quantity: integer`

In this schema the `componentid` field of Assembly is intended to be the `pid` of some part that is used as a component in assembling the part with `pid` equal to `partid`. Thus, the Assembly table represents a 1:N relationship between parts and their subparts; a part can have many subparts, but each part is the subpart of at most one part. In the Parts table `pid` is the key. For composite parts (those assembled from other parts, as indicated by the contents of Assembly), the `cost` field is taken to be the cost of assembling the part from its subparts.
Suppose that a frequent query is to find the (immediate) subparts of all parts that are supplied by a given supplier:

```sql
SELECT P.pid, A.componentid
FROM Parts P, Assembly A
WHERE P.pid = A.partid AND P.supplierid = 'Acme'
```

A good evaluation plan is to apply the selection condition on Parts and to then retrieve matching Assembly tuples through an index on the `partid` field. Ideally, the index on `partid` should be clustered. This plan is reasonably good. However, if such selections are common and we want to optimize them further, we can **co-cluster** the two tables. In this approach we store records of the two tables together, with each Parts record \( P \) followed by all the Assembly records \( A \) such that \( P.pid = A.partid \). This approach improves on storing the two relations separately and having a clustered index on `partid` because it doesn't need an index lookup to find the Assembly records that match a given Parts record. Thus, for each selection query, we save a few (typically two or three) index page I/Os.

If we are interested in finding the immediate subparts of all parts (i.e., the above query without the selection on `supplierid`), creating a clustered index on `partid` and doing an index nested loops join with Assembly as the inner relation offers good performance. An even better strategy is to create a clustered index on the `partid` field of Assembly and the `pid` field of Parts, and to then do a sort-merge join, using the indexes to retrieve tuples in sorted order. This strategy is comparable to doing the join using a co-clustered organization, which involves just one scan of the set of tuples (of Parts and Assembly, which are stored together in interleaved fashion).

The real benefit of co-clustering is illustrated by the following query:

```sql
SELECT P.pid, A.componentid
FROM Parts P, Assembly A
WHERE P.pid = A.partid AND P.cost=10
```

Suppose that many parts have `cost = 10`. This query essentially amounts to a collection of queries in which we are given a Parts record and want to find matching Assembly records. If we have an index on the `cost` field of Parts, we can retrieve qualifying Parts tuples. For each such tuple we have to use the index on Assembly to locate records with the given `pid`. The index access for Assembly is avoided if we have a co-clustered organization. (Of course, we still require an index on the `cost` attribute of Parts tuples.)

Such an optimization is especially important if we want to traverse several levels of the part-subpart hierarchy. For example, a common query is to find the total cost of a part, which requires us to repeatedly carry out joins of Parts and Assembly. Incidentally, if we don’t know the number of levels in the hierarchy in advance, the
number of joins varies and the query cannot be expressed in SQL. The query can
be answered by embedding an SQL statement for the join inside an iterative host
language program. How to express the query is orthogonal to our main point here,
which is that co-clustering is especially beneficial when the join in question is carried
out very frequently (either because it arises repeatedly in an important query such as
finding total cost, or because the join query is itself asked very frequently).

To summarize co-clustering:

- It can speed up joins, in particular key–foreign key joins corresponding to 1:N
  relationships.
- A sequential scan of either relation becomes slower. (In our example, since several
  Assembly tuples are stored in between consecutive Parts tuples, a scan of all
  Parts tuples becomes slower than if Parts tuples were stored separately. Similarly,
  a sequential scan of all Assembly tuples is also slower.)
- Inserts, deletes, and updates that alter record lengths all become slower, thanks
to the overheads involved in maintaining the clustering. (We will not discuss the
implementation issues involved in co-clustering.)

16.5 INDEXES ON MULTIPLE-ATTRIBUTE SEARCH KEYS *

It is sometimes best to build an index on a search key that contains more than one field.
For example, if we want to retrieve Employees records with $age = 30$ and $sal = 4000$, an
index with search key $\langle age, sal \rangle$ (or $\langle sal, age \rangle$) is superior to an index with search key
$age$ or an index with search key $sal$. If we have two indexes, one on $age$ and one on
$sal$, we could use them both to answer the query by retrieving and intersecting rids.
However, if we are considering what indexes to create for the sake of this query, we are
better off building one composite index.

Issues such as whether to make the index clustered or unclustered, dense or sparse, and
so on are orthogonal to the choice of the search key. We will call indexes on multiple-
attribute search keys composite indexes. In addition to supporting equality queries on
more than one attribute, composite indexes can be used to support multidimensional
range queries.

Consider the following query, which returns all employees with 20 < $age$ < 30 and
3000 < $sal$ < 5000:

```sql
SELECT E.eid
FROM Employees E
WHERE E.age BETWEEN 20 AND 30
     AND E.sal BETWEEN 3000 AND 5000
```
A composite index on \((age, sal)\) could help if the conditions in the \textbf{WHERE} clause are fairly selective. Obviously, a hash index will not help; a B+ tree (or ISAM) index is required. It is also clear that a clustered index is likely to be superior to an unclustered index. For this query, in which the conditions on \textit{age} and \textit{sal} are equally selective, a composite, clustered B+ tree index on \((age, sal)\) is as effective as a composite, clustered B+ tree index on \((sal, age)\). However, the order of search key attributes can sometimes make a big difference, as the next query illustrates:

```
SELECT E.eid
FROM Employees E
WHERE E.age = 25
    AND E.sal BETWEEN 3000 AND 5000
```

In this query a composite, clustered B+ tree index on \((age, sal)\) will give good performance because records are sorted by \textit{age} first and then (if two records have the same \textit{age} value) by \textit{sal}. Thus, all records with \textit{age} = 25 are clustered together. On the other hand, a composite, clustered B+ tree index on \((sal, age)\) will not perform as well. In this case, records are sorted by \textit{sal} first, and therefore two records with the same \textit{age} value (in particular, with \textit{age} = 25) may be quite far apart. In effect, this index allows us to use the range selection on \textit{sal}, but not the equality selection on \textit{age}, to retrieve tuples. (Good performance on both variants of the query can be achieved using a single spatial index. We discuss spatial indexes in Chapter 26.)

Some points about composite indexes are worth mentioning. Since data entries in the index contain more information about the data record (i.e., more fields than a single-attribute index), the opportunities for index-only evaluation strategies are increased (see Section 16.6). On the negative side, a composite index must be updated in response to any operation (insert, delete, or update) that modifies any field in the search key. A composite index is likely to be larger than a single-attribute search key index because the size of entries is larger. For a composite B+ tree index, this also means a potential increase in the number of levels, although key compression can be used to alleviate this problem (see Section 9.8.1).

### 16.6 INDEXES THAT ENABLE INDEX-ONLY PLANS *

This section considers a number of queries for which we can find efficient plans that avoid retrieving tuples from one of the referenced relations; instead, these plans scan an associated index (which is likely to be much smaller). An index that is used (only) for index-only scans does not have to be clustered because tuples from the indexed relation are not retrieved! However, only dense indexes can be used for the index-only strategies discussed here.

This query retrieves the managers of departments with at least one employee:
SELECT D.mgr
FROM  Departments D, Employees E
WHERE D.dno=E.dno

Observe that no attributes of Employees are retained. If we have a dense index on the 
dno field of Employees, the optimization of doing an index nested loops join using an
index-only scan for the inner relation is applicable; this optimization is discussed in
Section 14.7. Note that it does not matter whether this index is clustered because we
do not retrieve Employees tuples anyway. Given this variant of the query, the correct
decision is to build an unclustered, dense index on the dno field of Employees, rather
than a (dense or sparse) clustered index.

The next query takes this idea a step further:

SELECT D.mgr, E.eid
FROM  Departments D, Employees E
WHERE D.dno=E.dno

If we have an index on the dno field of Employees, we can use it to retrieve Employees
tuples during the join (with Departments as the outer relation), but unless the index
is clustered, this approach will not be efficient. On the other hand, suppose that we
have a dense B+ tree index on \langle dno, eid \rangle. Now all the information we need about an
Employees tuple is contained in the data entry for this tuple in the index. We can use
the index to find the first data entry with a given dno; all data entries with the same
dno are stored together in the index. (Note that a hash index on the composite key
\langle dno, eid \rangle cannot be used to locate an entry with just a given dno!) We can therefore
evaluate this query using an index nested loops join with Departments as the outer
relation and an index-only scan of the inner relation.

The next query shows how aggregate operations can influence the choice of indexes:

SELECT E.dno, COUNT(*)
FROM  Employees E
GROUP BY E.dno

A straightforward plan for this query is to sort Employees on dno in order to compute
the count of employees for each dno. However, if a dense index—hash or B+ tree—is
available, we can answer this query by scanning only the index. For each dno value,
we simply count the number of data entries in the index with this value for the search
key. Note that it does not matter whether the index is clustered because we never
retrieve tuples of Employees.

Here is a variation of the previous example:

SELECT E.dno, COUNT(*)
An index on \textit{dno} alone will not allow us to evaluate this query with an index-only scan, because we need to look at the \textit{sal} field of each tuple to verify that \textit{sal} = 10,000.

However, we can use an index-only plan if we have a composite B+ tree index on \langle \textit{sal}, \textit{dno} \rangle or \langle \textit{dno}, \textit{sal} \rangle. In an index with key \langle \textit{sal}, \textit{dno} \rangle, all data entries with \textit{sal} = 10,000 are arranged contiguously (whether or not the index is clustered). Further, these entries are sorted by \textit{dno}, making it easy to obtain a count for each \textit{dno} group. Note that we need to retrieve only data entries with \textit{sal} = 10,000. It is worth observing that this strategy will not work if the \textbf{WHERE} clause is modified to use \textit{sal} > 10,000. Although it suffices to retrieve only index data entries—that is, an index-only strategy still applies—these entries must now be sorted by \textit{dno} to identify the groups (because, for example, two entries with the same \textit{dno} but different \textit{sal} values may not be contiguous).

In an index with key \langle \textit{dno}, \textit{sal} \rangle, data entries with a given \textit{dno} value are stored together, and each such group of entries is itself sorted by \textit{sal}. For each \textit{dno} group, we can eliminate the entries with \textit{sal} not equal to 10,000 and count the rest. We observe that this strategy works even if the \textbf{WHERE} clause uses \textit{sal} > 10,000. Of course, this method is less efficient than an index-only scan with key \langle \textit{sal}, \textit{dno} \rangle because we must read all data entries.

As another example, suppose that we want to find the minimum \textit{sal} for each \textit{dno}:

\begin{verbatim}
SELECT E.dno, MIN(E.sal)
FROM Employees E
GROUP BY E.dno
\end{verbatim}

An index on \textit{dno} alone will not allow us to evaluate this query with an index-only scan. However, we can use an index-only plan if we have a composite B+ tree index on \langle \textit{dno}, \textit{sal} \rangle. Notice that all data entries in the index with a given \textit{dno} value are stored together (whether or not the index is clustered). Further, this group of entries is itself sorted by \textit{sal}. An index on \langle \textit{sal}, \textit{dno} \rangle would enable us to avoid retrieving data records, but the index data entries must be sorted on \textit{dno}.

Finally consider the following query:

\begin{verbatim}
SELECT AVG(E.sal)
FROM Employees E
WHERE E.age = 25
AND E.sal BETWEEN 3000 AND 5000
\end{verbatim}
A dense, composite B+ tree index on \((age, sal)\) allows us to answer the query with an index-only scan. A dense, composite B+ tree index on \((sal, age)\) will also allow us to answer the query with an index-only scan, although more index entries are retrieved in this case than with an index on \((age, sal)\).

### 16.7 OVERVIEW OF DATABASE TUNING

After the initial phase of database design, actual use of the database provides a valuable source of detailed information that can be used to refine the initial design. Many of the original assumptions about the expected workload can be replaced by observed usage patterns; in general, some of the initial workload specification will be validated, and some of it will turn out to be wrong. Initial guesses about the size of data can be replaced with actual statistics from the system catalogs (although this information will keep changing as the system evolves). Careful monitoring of queries can reveal unexpected problems; for example, the optimizer may not be using some indexes as intended to produce good plans.

Continued database tuning is important to get the best possible performance. In this section, we introduce three kinds of tuning: tuning indexes, tuning the conceptual schema, and tuning queries. Our discussion of index selection also applies to index tuning decisions. Conceptual schema and query tuning are discussed further in Sections 16.8 and 16.9.

#### 16.7.1 Tuning Indexes

The initial choice of indexes may be refined for one of several reasons. The simplest reason is that the observed workload reveals that some queries and updates considered important in the initial workload specification are not very frequent. The observed workload may also identify some new queries and updates that are important. The initial choice of indexes has to be reviewed in light of this new information. Some of the original indexes may be dropped and new ones added. The reasoning involved is similar to that used in the initial design.

It may also be discovered that the optimizer in a given system is not finding some of the plans that it was expected to. For example, consider the following query, which we discussed earlier:

```sql
SELECT D.mgr
FROM Employees E, Departments D
WHERE D.dname='Toy' AND E.dno=D.dno
```

A good plan here would be to use an index on `dname` to retrieve Departments tuples with `dname='Toy'` and to use a dense index on the `dno` field of Employees as the inner
relation, using an index-only scan. Anticipating that the optimizer would find such a plan, we might have created a dense, unclustered index on the \textit{dno} field of Employees.

Now suppose that queries of this form take an unexpectedly long time to execute. We can ask to see the plan produced by the optimizer. (Most commercial systems provide a simple command to do this.) If the plan indicates that an index-only scan is not being used, but that Employees tuples are being retrieved, we have to rethink our initial choice of index, given this revelation about our system’s (unfortunate) limitations. An alternative to consider here would be to drop the unclustered index on the \textit{dno} field of Employees and to replace it with a clustered index.

Some other common limitations of optimizers are that they do not handle selections involving string expressions, arithmetic, or \textit{null} values effectively. We discuss these points further when we consider query tuning in Section 16.9.

In addition to re-examining our choice of indexes, it pays to periodically reorganize some indexes. For example, a static index such as an ISAM index may have developed long overflow chains. Dropping the index and rebuilding it—if feasible, given the interrupted access to the indexed relation—can substantially improve access times through this index. Even for a dynamic structure such as a B+ tree, if the implementation does not merge pages on deletes, space occupancy can decrease considerably in some situations. This in turn makes the size of the index (in pages) larger than necessary, and could increase the height and therefore the access time. Rebuilding the index should be considered. Extensive updates to a clustered index might also lead to overflow pages being allocated, thereby decreasing the degree of clustering. Again, rebuilding the index may be worthwhile.

Finally, note that the query optimizer relies on statistics maintained in the system catalogs. These statistics are updated only when a special utility program is run; be sure to run the utility frequently enough to keep the statistics reasonably current.

### 16.7.2 Tuning the Conceptual Schema

In the course of database design, we may realize that our current choice of relation schemas does not enable us to meet our performance objectives for the given workload with any (feasible) set of physical design choices. If so, we may have to redesign our conceptual schema (and re-examine physical design decisions that are affected by the changes that we make).

We may realize that a redesign is necessary during the initial design process or later, after the system has been in use for a while. Once a database has been designed and populated with tuples, changing the conceptual schema requires a significant effort in terms of mapping the contents of relations that are affected. Nonetheless, it may
sometimes be necessary to revise the conceptual schema in light of experience with the
system. (Such changes to the schema of an operational system are sometimes referred
to as schema evolution.) We now consider the issues involved in conceptual schema
(re)design from the point of view of performance.

The main point to understand is that our choice of conceptual schema should be guided
by a consideration of the queries and updates in our workload, in addition to the issues
of redundancy that motivate normalization (which we discussed in Chapter 15). Several
options must be considered while tuning the conceptual schema:

- We may decide to settle for a 3NF design instead of a BCNF design.
- If there are two ways to decompose a given schema into 3NF or BCNF, our choice
  should be guided by the workload.
- Sometimes we might decide to further decompose a relation that is already
  in BCNF.
- In other situations we might denormalize. That is, we might choose to replace a
  collection of relations obtained by a decomposition from a larger relation with the
  original (larger) relation, even though it suffers from some redundancy problems.
  Alternatively, we might choose to add some fields to certain relations to speed
  up some important queries, even if this leads to a redundant storage of some
  information (and consequently, a schema that is in neither 3NF nor BCNF).
- This discussion of normalization has concentrated on the technique of decomposi-
tion, which amounts to vertical partitioning of a relation. Another technique to
consider is horizontal partitioning of a relation, which would lead to our having
two relations with identical schemas. Note that we are not talking about physi-

cally partitioning the tuples of a single relation; rather, we want to create two

Incidentally, when we redesign the conceptual schema, especially if we are tuning an
existing database schema, it is worth considering whether we should create views to
mask these changes from users for whom the original schema is more natural. We will
discuss the choices involved in tuning the conceptual schema in Section 16.8.

16.7.3 Tuning Queries and Views

If we notice that a query is running much slower than we expected, we have to examine
the query carefully to find the problem. Some rewriting of the query, perhaps in
conjunction with some index tuning, can often fix the problem. Similar tuning may
be called for if queries on some view run slower than expected. We will not discuss
view tuning separately; just think of queries on views as queries in their own right
When tuning a query, the first thing to verify is that the system is using the plan that you expect it to use. It may be that the system is not finding the best plan for a variety of reasons. Some common situations that are not handled efficiently by many optimizers follow.

- A selection condition involving null values.
- Selection conditions involving arithmetic or string expressions or conditions using the OR connective. For example, if we have a condition \( E.age = 2 \times D.age \) in the \texttt{WHERE} clause, the optimizer may correctly utilize an available index on \( E.age \) but fail to utilize an available index on \( D.age \). Replacing the condition by \( E.age/2 = D.age \) would reverse the situation.
- Inability to recognize a sophisticated plan such as an index-only scan for an aggregation query involving a \texttt{GROUP BY} clause. Of course, virtually no optimizer will look for plans outside the plan space described in Chapters 12 and 13, such as non-left-deep join trees. So a good understanding of what an optimizer typically does is important. In addition, the more aware you are of a given system’s strengths and limitations, the better off you are.

If the optimizer is not smart enough to find the best plan (using access methods and evaluation strategies supported by the DBMS), some systems allow users to guide the choice of a plan by providing hints to the optimizer; for example, users might be able to force the use of a particular index or choose the join order and join method. A user who wishes to guide optimization in this manner should have a thorough understanding of both optimization and the capabilities of the given DBMS. We will discuss query tuning further in Section 16.9.

16.8 CHOICES IN TUNING THE CONCEPTUAL SCHEMA *

We now illustrate the choices involved in tuning the conceptual schema through several examples using the following schemas:

- \texttt{Contracts\texttt{(cid: integer, supplierid: integer, projectid: integer,}
  
  \texttt{deptid: integer, partid: integer, qty: integer, value: real)}}
- \texttt{Departments\texttt{(did: integer, budget: real, annualreport: varchar)}}
- \texttt{Parts\texttt{(pid: integer, cost: integer)}}
- \texttt{Projects\texttt{(jid: integer, mgr: char(20)}})
- \texttt{Suppliers\texttt{(sid: integer, address: char(50)}})

For brevity, we will often use the common convention of denoting attributes by a single character and denoting relation schemas by a sequence of characters. Consider
the schema for the relation Contracts, which we will denote as CSJDPQV, with each letter denoting an attribute. The meaning of a tuple in this relation is that the contract with \(cid\) is an agreement that supplier \(S\) (with \(sid\) equal to \(supplierid\)) will supply \(Q\) items of part \(P\) (with \(pid\) equal to \(partid\)) to project \(J\) (with \(jid\) equal to \(projectid\)) associated with department \(D\) (with \(deptid\) equal to \(did\)), and that the value \(V\) of this contract is equal to \(value\).²

There are two known integrity constraints with respect to Contracts. A project purchases a given part using a single contract; thus, there will not be two distinct contracts in which the same project buys the same part. This constraint is represented using the FD \(JP \rightarrow C\). Also, a department purchases at most one part from any given supplier. This constraint is represented using the FD \(SD \rightarrow P\). In addition, of course, the contract id \(C\) is a key. The meaning of the other relations should be obvious, and we will not describe them further because our focus will be on the Contracts relation.

### 16.8.1 Settling for a Weaker Normal Form

Consider the Contracts relation. Should we decompose it into smaller relations? Let us see what normal form it is in. The candidate keys for this relation are \(C\) and \(JP\). (\(C\) is given to be a key, and \(JP\) functionally determines \(C\).) The only nonkey dependency is \(SD \rightarrow P\), and \(P\) is a prime attribute because it is part of candidate key \(JP\). Thus, the relation is not in BCNF—because there is a nonkey dependency—but it is in 3NF.

By using the dependency \(SD \rightarrow P\) to guide the decomposition, we get the two schemas \(SDP\) and \(CSJDPQV\). This decomposition is lossless, but it is not dependency-preserving. However, by adding the relation scheme \(CJP\), we obtain a lossless-join and dependency-preserving decomposition into BCNF. Using the guideline that a dependency-preserving, lossless-join decomposition into BCNF is good, we might decide to replace Contracts by three relations with schemas \(CJP\), \(SDP\), and \(CSJDPQV\).

However, suppose that the following query is very frequently asked: Find the number of copies \(Q\) of part \(P\) ordered in contract \(C\). This query requires a join of the decomposed relations \(CJP\) and \(CSJDPQV\) (or of \(SDP\) and \(CSJDPQV\)), whereas it can be answered directly using the relation Contracts. The added cost for this query could persuade us to settle for a 3NF design and not decompose Contracts further.

### 16.8.2 Denormalization

The reasons motivating us to settle for a weaker normal form may lead us to take an even more extreme step: deliberately introduce some redundancy. As an example,

²If this schema seems complicated, note that real-life situations often call for considerably more complex schemas!
consider the Contracts relation, which is in 3NF. Now, suppose that a frequent query
is to check that the value of a contract is less than the budget of the contracting
department. We might decide to add a budget field B to Contracts. Since did is a
key for Departments, we now have the dependency D → B in Contracts, which means
Contracts is not in 3NF any more. Nonetheless, we might choose to stay with this
design if the motivating query is sufficiently important. Such a decision is clearly
subjective and comes at the cost of significant redundancy.

16.8.3 Choice of Decompositions

Consider the Contracts relation again. Several choices are possible for dealing with
the redundancy in this relation:

- We can leave Contracts as it is and accept the redundancy associated with its
  being in 3NF rather than BCNF.
- We might decide that we want to avoid the anomalies resulting from this redun-
dancy by decomposing Contracts into BCNF using one of the following methods:
  - We have a lossless-join decomposition into PartInfo with attributes SDP and
    ContractInfo with attributes CSJDQV. As noted previously, this decompo-
sition is not dependency-preserving, and to make it dependency-preserving
    would require us to add a third relation CJP, whose sole purpose is to allow
    us to check the dependency JP → C.
  - We could choose to replace Contracts by just PartInfo and ContractInfo even
    though this decomposition is not dependency-preserving.

Replacing Contracts by just PartInfo and ContractInfo does not prevent us from en-
forcing the constraint JP → C; it only makes this more expensive. We could create
an assertion in SQL-92 to check this constraint:

```sql
CREATE ASSERTION checkDep
CHECK ( NOT EXISTS
( SELECT *
FROM PartInfo PI, ContractInfo CI
WHERE PI.supplierid=CI.supplierid
    AND PI.deptid=CI.deptid
GROUP BY CI.projectid, PI.partid
HAVING COUNT (cid) > 1 ) )
```

This assertion is expensive to evaluate because it involves a join followed by a sort
(to do the grouping). In comparison, the system can check that JP is a primary key
for table CJP by maintaining an index on JP. This difference in integrity-checking
cost is the motivation for dependency-preservation. On the other hand, if updates are
infrequent, this increased cost may be acceptable; therefore, we might choose not to
maintain the table CJP (and quite likely, an index on it).

As another example illustrating decomposition choices, consider the Contracts relation
again, and suppose that we also have the integrity constraint that a department uses
a given supplier for at most one of its projects: \( SPQ \rightarrow V \). Proceeding as before, we
have a lossless-join decomposition of Contracts into SDP and CSJDQV. Alternatively,
we could begin by using the dependency \( SPQ \rightarrow V \) to guide our decomposition, and
replace Contracts with SPQV and CSJDQPQ. We can then decompose CSJDQPQ, guided
by \( SD \rightarrow P \), to obtain SDP and CSJDQ.

Thus, we now have two alternative lossless-join decompositions of Contracts into
BCNF, neither of which is dependency-preserving. The first alternative is to replace
Contracts with the relations SDP and CSJDQV. The second alternative is to replace it
with SPQV, SDP, and CSJDQ. The addition of CJP makes the second decomposition
(but not the first!) dependency-preserving. Again, the cost of maintaining the three
relations CJP, SPQV, and CSJDQ (versus just CSJDQV) may lead us to choose the
first alternative. In this case, enforcing the given FDs becomes more expensive. We
might consider not enforcing them, but we then risk a violation of the integrity of our
data.

### 16.8.4 Vertical Decomposition

Suppose that we have decided to decompose Contracts into SDP and CSJDQV. These
schemas are in BCNF, and there is no reason to decompose them further from a nor-
malization standpoint. However, suppose that the following queries are very frequent:

- Find the contracts held by supplier S.
- Find the contracts placed by department D.

These queries might lead us to decompose CSJDQV into CS, CD, and CJQV. The
decomposition is lossless, of course, and the two important queries can be answered by
examining much smaller relations.

Whenever we decompose a relation, we have to consider which queries the decompo-
sition might adversely affect, especially if the only motivation for the decomposition
is improved performance. For example, if another important query is to find the to-
tal value of contracts held by a supplier, it would involve a join of the decomposed
relations CS and CJQV. In this situation we might decide against the decomposition.
16.8.5 Horizontal Decomposition

Thus far, we have essentially considered how to replace a relation with a collection of vertical decompositions. Sometimes, it is worth considering whether to replace a relation with two relations that have the same attributes as the original relation, each containing a subset of the tuples in the original. Intuitively, this technique is useful when different subsets of tuples are queried in very distinct ways.

For example, different rules may govern large contracts, which are defined as contracts with values greater than 10,000. (Perhaps such contracts have to be awarded through a bidding process.) This constraint could lead to a number of queries in which Contracts tuples are selected using a condition of the form \( value > 10,000 \). One way to approach this situation is to build a clustered B+ tree index on the \( value \) field of Contracts. Alternatively, we could replace Contracts with two relations called LargeContracts and SmallContracts, with the obvious meaning. If this query is the only motivation for the index, horizontal decomposition offers all the benefits of the index without the overhead of index maintenance. This alternative is especially attractive if other important queries on Contracts also require clustered indexes (on fields other than \( value \)).

If we replace Contracts by two relations LargeContracts and SmallContracts, we could mask this change by defining a view called Contracts:

```sql
CREATE VIEW Contracts(cid, supplierid, projectid, deptid, partid, qty, value)
AS ((SELECT *
    FROM LargeContracts)
UNION
(SELECT *
    FROM SmallContracts))
```

However, any query that deals solely with LargeContracts should be expressed directly on LargeContracts, and not on the view. Expressing the query on the view Contracts with the selection condition \( value > 10,000 \) is equivalent to expressing the query on LargeContracts, but less efficient. This point is quite general: Although we can mask changes to the conceptual schema by adding view definitions, users concerned about performance have to be aware of the change.

As another example, if Contracts had an additional field \( year \) and queries typically dealt with the contracts in some one year, we might choose to partition Contracts by year. Of course, queries that involved contracts from more than one year might require us to pose queries against each of the decomposed relations.
16.9 CHOICES IN TUNING QUERIES AND VIEWS *

The first step in tuning a query is to understand the plan that is used by the DBMS to evaluate the query. Systems usually provide some facility for identifying the plan used to evaluate a query. Once we understand the plan selected by the system, we can consider how to improve performance. We can consider a different choice of indexes or perhaps co-clustering two relations for join queries, guided by our understanding of the old plan and a better plan that we want the DBMS to use. The details are similar to the initial design process.

One point worth making is that before creating new indexes we should consider whether rewriting the query will achieve acceptable results with existing indexes. For example, consider the following query with an OR connective:

\[
\begin{align*}
\text{SELECT} & \quad \text{E.dno} \\
\text{FROM} & \quad \text{Employees E} \\
\text{WHERE} & \quad \text{E.hobby='Stamps'} \quad \text{OR} \quad \text{E.age}=10
\end{align*}
\]

If we have indexes on both hobby and age, we can use these indexes to retrieve the necessary tuples, but an optimizer might fail to recognize this opportunity. The optimizer might view the conditions in the WHERE clause as a whole as not matching either index, do a sequential scan of Employees, and apply the selections on-the-fly. Suppose we rewrite the query as the union of two queries, one with the clause \text{WHERE E.hobby='Stamps'} and the other with the clause \text{WHERE E.age}=10. Now each of these queries will be answered efficiently with the aid of the indexes on hobby and age.

We should also consider rewriting the query to avoid some expensive operations. For example, including \text{DISTINCT} in the SELECT clause leads to duplicate elimination, which can be costly. Thus, we should omit \text{DISTINCT} whenever possible. For example, for a query on a single relation, we can omit \text{DISTINCT} whenever either of the following conditions holds:

- We do not care about the presence of duplicates.
- The attributes mentioned in the SELECT clause include a candidate key for the relation.

Sometimes a query with \text{GROUP BY} and \text{HAVING} can be replaced by a query without these clauses, thereby eliminating a sort operation. For example, consider:

\[
\begin{align*}
\text{SELECT} & \quad \text{MIN (E.age)} \\
\text{FROM} & \quad \text{Employees E} \\
\text{GROUP BY} & \quad \text{E.dno} \\
\text{HAVING} & \quad \text{E.dno}=102
\end{align*}
\]
This query is equivalent to

```
SELECT  MIN (E.age)
FROM     Employees E
WHERE    E.dno=102
```

Complex queries are often written in steps, using a temporary relation. We can usually rewrite such queries without the temporary relation to make them run faster. Consider the following query for computing the average salary of departments managed by Robinson:

```
SELECT  * INTO Temp
FROM     Employees E, Departments D
WHERE    E.dno=D.dno AND D.mgrname='Robinson'
SELECT  T.dno, AVG (T.sal)
FROM     Temp T
GROUP BY T.dno
```

This query can be rewritten as

```
SELECT  E.dno, AVG (E.sal)
FROM     Employees E, Departments D
WHERE    E.dno=D.dno AND D.mgrname='Robinson'
GROUP BY E.dno
```

The rewritten query does not materialize the intermediate relation Temp and is therefore likely to be faster. In fact, the optimizer may even find a very efficient index-only plan that never retrieves Employees tuples if there is a dense, composite B+ tree index on `(dno, sal)`. This example illustrates a general observation: By rewriting queries to avoid unnecessary temporaries, we not only avoid creating the temporary relations, we also open up more optimization possibilities for the optimizer to explore.

In some situations, however, if the optimizer is unable to find a good plan for a complex query (typically a nested query with correlation), it may be worthwhile to rewrite the query using temporary relations to guide the optimizer toward a good plan.

In fact, nested queries are a common source of inefficiency because many optimizers deal poorly with them, as discussed in Section 14.5. Whenever possible, it is better to rewrite a nested query without nesting and to rewrite a correlated query without correlation. As already noted, a good reformulation of the query may require us to introduce new, temporary relations, and techniques to do so systematically (ideally, to
be done by the optimizer) have been widely studied. Often though, it is possible to
rewrite nested queries without nesting or the use of temporary relations, as illustrated in
Section 14.5.

16.10 IMPACT OF CONCURRENCY *

In a system with many concurrent users, several additional points must be considered.
As we saw in Chapter 1, each user’s program (transaction) obtains locks on the pages
that it reads or writes. Other transactions cannot access locked pages until this trans-
action completes and releases the locks. This restriction can lead to contention for
locks on heavily used pages.

- The duration for which transactions hold locks can affect performance signifi-
cantly. Tuning transactions by writing to local program variables and deferring
changes to the database until the end of the transaction (and thereby delaying the
acquisition of the corresponding locks) can greatly improve performance. On a
related note, performance can be improved by replacing a transaction with several
smaller transactions, each of which holds locks for a shorter time.

- At the physical level, a careful partitioning of the tuples in a relation and its
associated indexes across a collection of disks can significantly improve concurrent
access. For example, if we have the relation on one disk and an index on another,
accesses to the index can proceed without interfering with accesses to the relation,
at least at the level of disk reads.

- If a relation is updated frequently, B+ tree indexes in particular can become a con-
currency control bottleneck because all accesses through the index must go through
the root; thus, the root and index pages just below it can become hotspots, that is,
pages for which there is heavy contention. If the DBMS uses specialized locking
protocols for tree indexes, and in particular, sets fine-granularity locks, this prob-
lem is greatly alleviated. Many current systems use such techniques. Nonetheless,
this consideration may lead us to choose an ISAM index in some situations. Be-
cause the index levels of an ISAM index are static, we do not need to obtain locks
on these pages; only the leaf pages need to be locked. An ISAM index may be
preferable to a B+ tree index, for example, if frequent updates occur but we ex-
pect the relative distribution of records and the number (and size) of records with
a given range of search key values to stay approximately the same. In this case the
ISAM index offers a lower locking overhead (and reduced contention for locks),
and the distribution of records is such that few overflow pages will be created.

Hashed indexes do not create such a concurrency bottleneck, unless the data
distribution is very skewed and many data items are concentrated in a few buckets.
In this case the directory entries for these buckets can become a hotspot.

- The pattern of updates to a relation can also become significant. For example,
if tuples are inserted into the Employees relation in eid order and we have a B+
A tree index on \textit{eid}, each insert will go to the last leaf page of the B+ tree. This leads to hotspots along the path from the root to the right-most leaf page. Such considerations may lead us to choose a hash index over a B+ tree index or to index on a different field. (Note that this pattern of access leads to poor performance for ISAM indexes as well, since the last leaf page becomes a hot spot.) Again, this is not a problem for hash indexes because the hashing process randomizes the bucket into which a record is inserted.

- SQL features for specifying transaction properties, which we discuss in Section 19.4, can be used for improving performance. If a transaction does not modify the database, we should specify that its \textit{access mode} is \textit{READ ONLY}. Sometimes it is acceptable for a transaction (e.g., one that computes statistical summaries) to see some anomalous data due to concurrent execution. For such transactions, more concurrency can be achieved by controlling a parameter called the \textit{isolation level}.

\section*{16.11 DBMS BENCHMARKING *}

Thus far, we have considered how to improve the design of a database to obtain better performance. As the database grows, however, the underlying DBMS may no longer be able to provide adequate performance even with the best possible design, and we have to consider upgrading our system, typically by buying faster hardware and additional memory. We may also consider migrating our database to a new DBMS.

When evaluating DBMS products, performance is an important consideration. A DBMS is a complex piece of software, and different vendors may target their systems toward different market segments by putting more effort into optimizing certain parts of the system, or by choosing different system designs. For example, some systems are designed to run complex queries efficiently, while others are designed to run many simple transactions per second. Within each category of systems, there are many competing products. To assist users in choosing a DBMS that is well suited to their needs, several \textbf{performance benchmarks} have been developed. These include benchmarks for measuring the performance of a certain class of applications (e.g., the TPC benchmarks) and benchmarks for measuring how well a DBMS performs various operations (e.g., the Wisconsin benchmark).

Benchmarks should be portable, easy to understand, and scale naturally to larger problem instances. They should measure \textit{peak performance} (e.g., \textit{transactions per second}, or \textit{tps}) as well as \textit{price/performance ratios} (e.g., $/\text{tps}$) for typical workloads in a given application domain. The Transaction Processing Council (TPC) was created to define benchmarks for transaction processing and database systems. Other well-known benchmarks have been proposed by academic researchers and industry organizations. Benchmarks that are proprietary to a given vendor are not very useful for comparing
different systems (although they may be useful in determining how well a given system would handle a particular workload).

### 16.11.1 Well-Known DBMS Benchmarks

**On-line Transaction Processing Benchmarks:** The TPC-A and TPC-B benchmarks constitute the standard definitions of the $tps$ and $$/tps$ measures. TPC-A measures the performance and price of a computer network in addition to the DBMS, whereas the TPC-B benchmark considers the DBMS by itself. These benchmarks involve a simple transaction that updates three data records, from three different tables, and appends a record to a fourth table. A number of details (e.g., transaction arrival distribution, interconnect method, system properties) are rigorously specified, ensuring that results for different systems can be meaningfully compared. The TPC-C benchmark is a more complex suite of transactional tasks than TPC-A and TPC-B. It models a warehouse that tracks items supplied to customers and involves five types of transactions. Each TPC-C transaction is much more expensive than a TPC-A or TPC-B transaction, and TPC-C exercises a much wider range of system capabilities, such as use of secondary indexes and transaction aborts. It has more or less completely replaced TPC-A and TPC-B as the standard transaction processing benchmark.

**Query Benchmarks:** The Wisconsin benchmark is widely used for measuring the performance of simple relational queries. The Set Query benchmark measures the performance of a suite of more complex queries, and the $AS^3AP$ benchmark measures the performance of a mixed workload of transactions, relational queries, and utility functions. The TPC-D benchmark is a suite of complex SQL queries, intended to be representative of the decision-support application domain. The OLAP Council has also developed a benchmark for complex decision-support queries, including some queries that cannot be expressed easily in SQL; this is intended to measure systems for on-line analytic processing (OLAP), which we discuss in Chapter 23, rather than traditional SQL systems. The Sequoia 2000 benchmark is designed to compare DBMS support for geographic information systems.

**Object-Database Benchmarks:** The 001 and 007 benchmarks measure the performance of object-oriented database systems. The Bucky benchmark measures the performance of object-relational database systems. (We discuss object database systems in Chapter 25.)

### 16.11.2 Using a Benchmark

Benchmarks should be used with a good understanding of what they are designed to measure and the application environment in which a DBMS is to be used. When you
use benchmarks to guide your choice of a DBMS, keep the following guidelines in mind:

- **How meaningful is a given benchmark?** Benchmarks that try to distill performance into a single number can be overly simplistic. A DBMS is a complex piece of software used in a variety of applications. A good benchmark should have a suite of tasks that are carefully chosen to cover a particular application domain and to test DBMS features that are important for that domain.

- **How well does a benchmark reflect your workload?** You should consider your expected workload and compare it with the benchmark. Give more weight to the performance of those benchmark tasks (i.e., queries and updates) that are similar to important tasks in your workload. Also consider how benchmark numbers are measured. For example, elapsed times for individual queries might be misleading if considered in a multiuser setting: A system may have higher elapsed times because of slower I/O. On a multiuser workload, given sufficient disks for parallel I/O, such a system might outperform a system with a lower elapsed time.

- **Create your own benchmark:** Vendors often tweak their systems in ad hoc ways to obtain good numbers on important benchmarks. To counter this, create your own benchmark by modifying standard benchmarks slightly or by replacing the tasks in a standard benchmark with similar tasks from your workload.

16.12 POINTS TO REVIEW

- In **physical design**, we adjust the physical schema according to the typical query and update workload. A *workload description* contains detailed information about queries, updates, and their frequencies. During physical design, we might create indexes, make changes to the conceptual schema, and/or rewrite queries. (Section 16.1)

- There are guidelines that help us to decide whether to index, what to index, whether to use a multiple-attribute index, whether to create an unclustered or a clustered index, and whether to use a hash or a tree index. Indexes can speed up queries but can also slow down update operations. (Section 16.2)

- When choosing indexes, we must consider complete query plans including potential join methods that benefit from the indexes. It is not enough to just consider the conditions in the *WHERE* clause as selection criteria for accessing individual relations. (Section 16.3)

- Range queries can benefit from clustered indexes. When deciding which index to create, we have to take the selectivity of conditions in the *WHERE* clause into account. Some systems allow us to store records from more than one relation in a single file. This physical layout, called *co-clustering*, can speed up key–foreign key joins, which arise frequently in practice. (Section 16.4)
If the WHERE condition has several conjunctions involving different attributes, an index on a search key with more than one field, called a composite index, can improve query performance. (Section 16.5)

Query plans that do not have to retrieve records from an underlying relation are called index-only plans. Indexes that are used for index-only access do not need to be clustered. (Section 16.6)

After an initial physical design, continuous database tuning is important to obtain best possible performance. Using the observed workload over time, we can reconsider our choice of indexes and our relation schema. Other tasks include periodic reorganization of indexes and updating the statistics in the system catalogs. (Section 16.7)

We can tune the conceptual schema for increased performance by settling for a weaker normal form or denormalizing a relation to speed up some important query. Usually, we have several decomposition choices that we need to investigate carefully. In some cases we can increase performance through vertical or horizontal decomposition of a relation. (Section 16.8)

When tuning queries, we first need to understand the query plan that the DBMS generates. Sometimes, query performance can be improved by rewriting the query in order to help the DBMS find a better query plan. (Section 16.9)

If many users access the database concurrently, lock contention can decrease performance. Several possibilities exist for decreasing concurrency bottlenecks. (Section 16.10)

A DBMS benchmark tests the performance of a class of applications or specific aspects of a DBMS to help users evaluate system performance. Well-known benchmarks include TPC-A, TPC-B, TPC-C, and TPC-D. (Section 16.11)

EXERCISES

Exercise 16.1 Consider the following relations:

```
Emp(eid: integer, ename: varchar, sal: integer, age: integer, did: integer)
Dept(did: integer, budget: integer, floor: integer, mgr_eid: integer)
```

Salaries range from $10,000 to $100,000, ages vary from 20 to 80, each department has about five employees on average, there are 10 floors, and budgets vary from $10,000 to $1,000,000. You can assume uniform distributions of values.

For each of the following queries, which of the listed index choices would you choose to speed up the query? If your database system does not consider index-only plans (i.e., data records are always retrieved even if enough information is available in the index entry), how would your answer change? Explain briefly.
1. Query: Print ename, age, and sal for all employees.
   (a) Clustered, dense hash index on \(<ename, age, sal>\) fields of Emp.
   (b) Unclustered hash index on \(<ename, age, sal>\) fields of Emp.
   (c) Clustered, sparse B+ tree index on \(<ename, age, sal>\) fields of Emp.
   (d) Unclustered hash index on \(<eid, did>\) fields of Emp.
   (e) No index.

2. Query: Find the dids of departments that are on the 10th floor and that have a budget of less than $15,000.
   (a) Clustered, dense hash index on the floor field of Dept.
   (b) Unclustered hash index on the floor field of Dept.
   (c) Clustered, dense B+ tree index on \(<floor, budget>\) fields of Dept.
   (d) Clustered, sparse B+ tree index on the budget field of Dept.
   (e) No index.

3. Query: Find the names of employees who manage some department and have a salary greater than $12,000.
   (a) Clustered, sparse B+ tree index on the sal field of Emp.
   (b) Clustered hash index on the did field of Dept.
   (c) Unclustered hash index on the did field of Dept.
   (d) Unclustered hash index on the did field of Emp.
   (e) Clustered B+ tree index on sal field of Emp and clustered hash index on the did field of Dept.

4. Query: Print the average salary for each department.
   (a) Clustered, sparse B+ tree index on the did field of Emp.
   (b) Clustered, dense B+ tree index on the did field of Emp.
   (c) Clustered, dense B+ tree index on \(<did, sal>\) fields of Emp.
   (d) Unclustered hash index on \(<did, sal>\) fields of Emp.
   (e) Clustered, dense B+ tree index on the did field of Dept.

Exercise 16.2 Consider the following relation:

\[\text{Emp}(\text{eid: integer, sal: integer, age: real, did: integer})\]

There is a clustered index on eid and an unclustered index on age.

1. Which factors would you consider in deciding whether to make an index on a relation a clustered index? Would you always create at least one clustered index on every relation?
2. How would you use the indexes to enforce the constraint that eid is a key?
3. Give an example of an update that is definitely speeded up because of the available indexes. (English description is sufficient.)
4. Give an example of an update that is definitely slowed down because of the indexes.
   (English description is sufficient.)

5. Can you give an example of an update that is neither speeded up nor slowed down by
   the indexes?

Exercise 16.3 Consider the following BCNF schema for a portion of a simple corporate
   database (type information is not relevant to this question and is omitted):

   \[
   \begin{align*}
   \text{Emp} & \quad (eid, \text{ename}, \text{addr}, \text{sal}, \text{age}, \text{yrs}, \text{deptid}) \\
   \text{Dept} & \quad (did, \text{dname}, \text{floor}, \text{budget})
   \end{align*}
   \]

Suppose you know that the following queries are the six most common queries in the workload
   for this corporation and that all six are roughly equivalent in frequency and importance:

- List the id, name, and address of employees in a user-specified age range.
- List the id, name, and address of employees who work in the department with a user-
  specified department name.
- List the id and address of employees with a user-specified employee name.
- List the overall average salary for employees.
- List the average salary for employees of each age; that is, for each age in the database, list
  the age and the corresponding average salary.
- List all the department information, ordered by department floor numbers.

1. Given this information, and assuming that these queries are more important than any
   updates, design a physical schema for the corporate database that will give good perfor-
   mance for the expected workload. In particular, decide which attributes will be indexed
   and whether each index will be a clustered index or an unclustered index. Assume that
   B+ tree indexes are the only index type supported by the DBMS and that both single-
   and multiple-attribute keys are permitted. Specify your physical design by identifying
   the attributes that you recommend indexing on via clustered or unclustered B+ trees.

2. Redesign the physical schema assuming that the set of important queries is changed to
   be the following:

   - List the id and address of employees with a user-specified employee name.
   - List the overall maximum salary for employees.
   - List the average salary for employees by department; that is, for each deptid value,
     list the deptid value and the average salary of employees in that department.
   - List the sum of the budgets of all departments by floor; that is, for each floor, list
     the floor and the sum.

Exercise 16.4 Consider the following BCNF relational schema for a portion of a university
   database (type information is not relevant to this question and is omitted):

   \[
   \begin{align*}
   \text{Prof} & \quad (ssno, \text{pname}, \text{office}, \text{age}, \text{sex}, \text{specialty}, \text{dept}\_did) \\
   \text{Dept} & \quad (did, \text{dname}, \text{budget}, \text{num}\_majors, \text{chair}\_ssno)
   \end{align*}
   \]
Suppose you know that the following queries are the five most common queries in the workload for this university and that all five are roughly equivalent in frequency and importance:

- List the names, ages, and offices of professors of a user-specified sex (male or female) who have a user-specified research specialty (e.g., recursive query processing). Assume that the university has a diverse set of faculty members, making it very uncommon for more than a few professors to have the same research specialty.
- List all the department information for departments with professors in a user-specified age range.
- List the department id, department name, and chairperson name for departments with a user-specified number of majors.
- List the lowest budget for a department in the university.
- List all the information about professors who are department chairpersons.

These queries occur much more frequently than updates, so you should build whatever indexes you need to speed up these queries. However, you should not build any unnecessary indexes, as updates will occur (and would be slowed down by unnecessary indexes). Given this information, design a physical schema for the university database that will give good performance for the expected workload. In particular, decide which attributes should be indexed and whether each index should be a clustered index or an unclustered index. Assume that both B+ trees and hashed indexes are supported by the DBMS and that both single- and multiple-attribute index search keys are permitted.

1. Specify your physical design by identifying the attributes that you recommend indexing on, indicating whether each index should be clustered or unclustered and whether it should be a B+ tree or a hashed index.
2. Redesign the physical schema assuming that the set of important queries is changed to be the following:
   - List the number of different specialties covered by professors in each department, by department.
   - Find the department with the fewest majors.
   - Find the youngest professor who is a department chairperson.

Exercise 16.5 Consider the following BCNF relational schema for a portion of a company database (type information is not relevant to this question and is omitted):

\[
\text{Project}(pno, proj\_name, proj\_base\_dept, proj\_mgr, topic, budget) \\
\text{Manager}(mid, mgr\_name, mgr\_dept, salary, age, sex)
\]

Note that each project is based in some department, each manager is employed in some department, and the manager of a project need not be employed in the same department (in which the project is based). Suppose you know that the following queries are the five most common queries in the workload for this university and that all five are roughly equivalent in frequency and importance:

- List the names, ages, and salaries of managers of a user-specified sex (male or female) working in a given department. You can assume that while there are many departments, each department contains very few project managers.
List the names of all projects with managers whose ages are in a user-specified range (e.g., younger than 30).

List the names of all departments such that a manager in this department manages a project based in this department.

List the name of the project with the lowest budget.

List the names of all managers in the same department as a given project.

These queries occur much more frequently than updates, so you should build whatever indexes you need to speed up these queries. However, you should not build any unnecessary indexes, as updates will occur (and would be slowed down by unnecessary indexes). Given this information, design a physical schema for the company database that will give good performance for the expected workload. In particular, decide which attributes should be indexed and whether each index should be a clustered index or an unclustered index. Assume that both B+ trees and hashed indexes are supported by the DBMS, and that both single- and multiple-attribute index keys are permitted.

1. Specify your physical design by identifying the attributes that you recommend indexing on, indicating whether each index should be clustered or unclustered and whether it should be a B+ tree or a hashed index.

2. Redesign the physical schema assuming that the set of important queries is changed to be the following:
   - Find the total of the budgets for projects managed by each manager; that is, list proj mgr and the total of the budgets of projects managed by that manager, for all values of proj mgr.
   - Find the total of the budgets for projects managed by each manager but only for managers who are in a user-specified age range.
   - Find the number of male managers.
   - Find the average age of managers.

Exercise 16.6 The Globetrotters Club is organized into chapters. The president of a chapter can never serve as the president of any other chapter, and each chapter gives its president some salary. Chapters keep moving to new locations, and a new president is elected when (and only when) a chapter moves. The above data is stored in a relation G(C,S,L,P), where the attributes are chapters (C), salaries (S), locations (L), and presidents (P). Queries of the following form are frequently asked, and you must be able to answer them without computing a join: “Who was the president of chapter X when it was in location Y?”

1. List the FDs that are given to hold over G.
2. What are the candidate keys for relation G?
3. What normal form is the schema G in?
4. Design a good database schema for the club. (Remember that your design must satisfy the query requirement stated above!)
5. What normal form is your good schema in? Give an example of a query that is likely to run slower on this schema than on the relation G.
6. Is there a lossless-join, dependency-preserving decomposition of G into BCNF?
7. Is there ever a good reason to accept something less than 3NF when designing a schema for a relational database? Use this example, if necessary adding further constraints, to illustrate your answer.

**Exercise 16.7** Consider the following BCNF relation, which lists the ids, types (e.g., nuts or bolts), and costs of various parts, along with the number that are available or in stock:

\[ \text{Parts (pid, pname, cost, num\_avail)} \]

You are told that the following two queries are extremely important:

- Find the total number available by part type, for all types. (That is, the sum of the num\_avail value of all nuts, the sum of the num\_avail value of all bolts, etc.)
- List the pids of parts with the highest cost.

1. Describe the physical design that you would choose for this relation. That is, what kind of a file structure would you choose for the set of Parts records, and what indexes would you create?
2. Suppose that your customers subsequently complain that performance is still not satisfactory (given the indexes and file organization that you chose for the Parts relation in response to the previous question). Since you cannot afford to buy new hardware or software, you have to consider a schema redesign. Explain how you would try to obtain better performance by describing the schema for the relation(s) that you would use and your choice of file organizations and indexes on these relations.
3. How would your answers to the above two questions change, if at all, if your system did not support indexes with multiple-attribute search keys?

**Exercise 16.8** Consider the following BCNF relations, which describe employees and departments that they work in:

\[ \text{Emp (eid, sal, did)} \]
\[ \text{Dept (did, location, budget)} \]

You are told that the following queries are extremely important:

- Find the location where a user-specified employee works.
- Check whether the budget of a department is greater than the salary of each employee in that department.

1. Describe the physical design that you would choose for this relation. That is, what kind of a file structure would you choose for these relations, and what indexes would you create?
2. Suppose that your customers subsequently complain that performance is still not satisfactory (given the indexes and file organization that you chose for the relations in response to the previous question). Since you cannot afford to buy new hardware or software, you have to consider a schema redesign. Explain how you would try to obtain better performance by describing the schema for the relation(s) that you would use and your choice of file organizations and indexes on these relations.
3. Suppose that your database system has very inefficient implementations of index structures. What kind of a design would you try in this case?

**Exercise 16.9** Consider the following BCNF relations, which describe departments in a company and employees:

\[
\text{Dept}(\text{did, dname, location, managerid}) \\
\text{Emp(vid, sal)}
\]

You are told that the following queries are extremely important:

- List the names and ids of managers for each department in a user-specified location, in alphabetical order by department name.
- Find the average salary of employees who manage departments in a user-specified location. You can assume that no one manages more than one department.

1. Describe the file structures and indexes that you would choose.
2. You subsequently realize that updates to these relations are frequent. Because indexes incur a high overhead, can you think of a way to improve performance on these queries without using indexes?

**Exercise 16.10** For each of the following queries, identify one possible reason why an optimizer might not find a good plan. Rewrite the query so that a good plan is likely to be found. Any available indexes or known constraints are listed before each query; assume that the relation schemas are consistent with the attributes referred to in the query.

1. An index is available on the \textit{age} attribute.
   \[
   \text{SELECT E.dno} \\
   \text{FROM Employee E} \\
   \text{WHERE E.age=20 OR E.age=10}
   \]

2. A B+ tree index is available on the \textit{age} attribute.
   \[
   \text{SELECT E.dno} \\
   \text{FROM Employee E} \\
   \text{WHERE E.age<20 AND E.age>10}
   \]

3. An index is available on the \textit{age} attribute.
   \[
   \text{SELECT E.dno} \\
   \text{FROM Employee E} \\
   \text{WHERE 2*E.age<20}
   \]

4. No indexes are available.
   \[
   \text{SELECT DISTINCT *} \\
   \text{FROM Employee E}
   \]

5. No indexes are available.
   \[
   \text{SELECT AVG (E.sal)} \\
   \text{FROM Employee E} \\
   \text{GROUP BY E.dno} \\
   \text{HAVING E.dno=22}
   \]
6. *sid* in Reserves is a foreign key that refers to Sailors.

```sql
SELECT S.sid 
FROM Sailors S, Reserves R 
WHERE S.sid = R.sid
```

**Exercise 16.11** Consider the following two ways of computing the names of employees who earn more than $100,000 and whose age is equal to their manager’s age. First, a nested query:

```sql
SELECT E1.ename 
FROM Emp E1 
WHERE E1.sal > 100 AND E1.age = ( 
  SELECT E2.age 
  FROM Emp E2, Dept D2 
  WHERE E1.dname = D2.dname 
  AND D2.mgr = E2.ename 
)
```

Second, a query that uses a view definition:

```sql
SELECT E1.ename 
FROM Emp E1, MgrAge A 
WHERE E1.dname = A.dname AND E1.sal > 100 AND E1.age = A.age
```

```sql
CREATE VIEW MgrAge (dname, age) 
AS SELECT D.dname, E.age 
FROM Emp E, Dept D 
WHERE D.mgr = E.ename
```

1. Describe a situation in which the first query is likely to outperform the second query.
2. Describe a situation in which the second query is likely to outperform the first query.
3. Can you construct an equivalent query that is likely to beat both these queries when every employee who earns more than $100,000 is either 35 or 40 years old? Explain briefly.

**PROJECT-BASED EXERCISES**

**Exercise 16.12** Minibase’s Designview tool does not provide any support for choosing indexes or, in general, physical database design. How do you see Designview being used, if at all, in the context of physical database design?

**BIBLIOGRAPHIC NOTES**

[572] is an early discussion of physical database design. [573] discusses the performance implications of normalization and observes that denormalization may improve performance for certain queries. The ideas underlying a physical design tool from IBM are described in
The Microsoft AutoAdmin tool that performs automatic index selection according to a query workload is described in [138]. Other approaches to physical database design are described in [125, 557]. [591] considers transaction tuning, which we discussed only briefly. The issue is how an application should be structured into a collection of transactions to maximize performance.

The following books on database design cover physical design issues in detail; they are recommended for further reading. [236] is largely independent of specific products, although many examples are based on DB2 and Teradata systems. [684] deals primarily with DB2. [589] is a very readable treatment of performance tuning and is not specific to any one system.

[284] contains several papers on benchmarking database systems and has accompanying software. It includes articles on the ASAP, Set Query, TPC-A, TPC-B, Wisconsin, and 001 benchmarks written by the original developers. The Bucky benchmark is described in [112], the 007 benchmark is described in [111], and the TPC-D benchmark is described in [648]. The Sequoia 2000 benchmark is described in [631].
I know that’s a secret, for it’s whispered everywhere.

— William Congreve

Database management systems are increasingly being used to store information about all aspects of an enterprise. The data stored in a DBMS is often vital to the business interests of the organization and is regarded as a corporate asset. In addition to protecting the intrinsic value of the data, corporations must consider ways to ensure privacy and to control access to data that must not be revealed to certain groups of users for various reasons.

In this chapter we discuss the concepts underlying access control and security in a DBMS. After introducing database security issues in Section 17.1, we consider two distinct approaches, called discretionary and mandatory, to specifying and managing access controls. An access control mechanism is a way to control the data that is accessible to a given user. After introducing access controls in Section 17.2 we cover discretionary access control, which is supported in SQL-92, in Section 17.3. We briefly cover mandatory access control, which is not supported in SQL-92, in Section 17.4.

In Section 17.5 we discuss several additional aspects of security, such as security in a statistical database, the role of the database administrator, and the use of techniques such as encryption and audit trails.

### 17.1 INTRODUCTION TO DATABASE SECURITY

There are three main objectives to consider while designing a secure database application:

1. **Secrecy:** Information should not be disclosed to unauthorized users. For example, a student should not be allowed to examine other students’ grades.

2. **Integrity:** Only authorized users should be allowed to modify data. For example, students may be allowed to see their grades, yet not allowed (obviously!) to modify them.

3. **Availability:** Authorized users should not be denied access. For example, an instructor who wishes to change a grade should be allowed to do so.
To achieve these objectives, a clear and consistent security policy should be developed to describe what security measures must be enforced. In particular, we must determine what part of the data is to be protected and which users get access to which portions of the data. Next, the security mechanisms of the underlying DBMS (and OS, as well as external mechanisms such as securing access to buildings and so on) must be utilized to enforce the policy. We emphasize that security measures must be taken at several levels. Security leaks in the operating system or network connections can circumvent database security mechanisms. For example, such leaks could allow an intruder to log on as the database administrator with all the attendant DBMS access rights! Human factors are another source of security leaks. For example, a user may choose a password that is easy to guess, or a user who is authorized to see sensitive data may misuse it. Such errors in fact account for a large percentage of security breaches. We will not discuss these aspects of security despite their importance because they are not specific to database management systems.

Views provide a valuable tool in enforcing security policies. The view mechanism can be used to create a ‘window’ on a collection of data that is appropriate for some group of users. Views allow us to limit access to sensitive data by providing access to a restricted version (defined through a view) of that data, rather than to the data itself.

We use the following schemas in our examples:

Sailors(sid: integer, sname: string, rating: integer, age: real)
Boats(bid: integer, bname: string, color: string)
Reserves(sname: string, bid: integer, day: dates)

Notice that Reserves has been modified to use sname, rather than sid.

17.2 ACCESS CONTROL

A database for an enterprise contains a great deal of information and usually has several groups of users. Most users need to access only a small part of the database to carry out their tasks. Allowing users unrestricted access to all the data can be undesirable, and a DBMS should provide mechanisms to control access to data.

A DBMS offers two main approaches to access control. Discretionary access control is based on the concept of access rights, or privileges, and mechanisms for giving users such privileges. A privilege allows a user to access some data object in a certain manner (e.g., to read or to modify). A user who creates a database object such as a table or a view automatically gets all applicable privileges on that object. The DBMS subsequently keeps track of how these privileges are granted to other users, and possibly revoked, and ensures that at all times only users with the necessary privileges can access an object. SQL-92 supports discretionary access control through
the GRANT and REVOKE commands. The GRANT command gives privileges to users, and the REVOKE command takes away privileges. We discuss discretionary access control in Section 17.3.

Discretionary access control mechanisms, while generally effective, have certain weaknesses. In particular, a devious unauthorized user can trick an authorized user into disclosing sensitive data. Mandatory access control is based on systemwide policies that cannot be changed by individual users. In this approach each database object is assigned a security class, each user is assigned clearance for a security class, and rules are imposed on reading and writing of database objects by users. The DBMS determines whether a given user can read or write a given object based on certain rules that involve the security level of the object and the clearance of the user. These rules seek to ensure that sensitive data can never be 'passed on' to a user without the necessary clearance. The SQL-92 standard does not include any support for mandatory access control. We discuss mandatory access control in Section 17.4.

17.3 DISCRETIONARY ACCESS CONTROL

SQL-92 supports discretionary access control through the GRANT and REVOKE commands. The GRANT command gives users privileges to base tables and views. The syntax of this command is as follows:

\[
\text{GRANT privileges ON object TO users [ WITH GRANT OPTION ]}
\]

For our purposes object is either a base table or a view. SQL recognizes certain other kinds of objects, but we will not discuss them. Several privileges can be specified, including these:

- **SELECT**: The right to access (read) all columns of the table specified as the object, including columns added later through ALTER TABLE commands.

- **INSERT(column-name)**: The right to insert rows with (non-null or nondefault) values in the named column of the table named as object. If this right is to be granted with respect to all columns, including columns that might be added later, we can simply use INSERT. The privileges UPDATE(column-name) and UPDATE are similar.

- **DELETE**: The right to delete rows from the table named as object.

- **REFERENCES(column-name)**: The right to define foreign keys (in other tables) that refer to the specified column of the table object. REFERENCES without a column name specified denotes this right with respect to all columns, including any that are added later.
If a user has a privilege with the grant option, he or she can pass it to another user (with or without the grant option) by using the GRANT command. A user who creates a base table automatically has all applicable privileges on it, along with the right to grant these privileges to other users. A user who creates a view has precisely those privileges on the view that he or she has on every one of the view or base tables used to define the view. The user creating the view must have the SELECT privilege on each underlying table, of course, and so is always granted the SELECT privilege on the view. The creator of the view has the SELECT privilege with the grant option only if he or she has the SELECT privilege with the grant option on every underlying table. In addition, if the view is updatable and the user holds INSERT, DELETE, or UPDATE privileges (with or without the grant option) on the (single) underlying table, the user automatically gets the same privileges on the view.

Only the owner of a schema can execute the data definition statements CREATE, ALTER, and DROP on that schema. The right to execute these statements cannot be granted or revoked.

In conjunction with the GRANT and REVOKE commands, views are an important component of the security mechanisms provided by a relational DBMS. By defining views on the base tables, we can present needed information to a user while hiding other information that the user should not be given access to. For example, consider the following view definition:

```
CREATE VIEW ActiveSailors (name, age, day)
AS SELECT S.sname, S.age, R.day
FROM Sailors S, Reserves R
WHERE S.sname = R.sname AND S.rating > 6
```

A user who can access ActiveSailors, but not Sailors or Reserves, knows which sailors have reservations but cannot find out the bids of boats reserved by a given sailor.

Privileges are assigned in SQL-92 to authorization ids, which can denote a single user or a group of users; a user must specify an authorization id and, in many systems, a corresponding password before the DBMS accepts any commands from him or her. So, technically, Joe, Michael, and so on are authorization ids rather than user names in the following examples.

Suppose that user Joe has created the tables Boats, Reserves, and Sailors. Some examples of the GRANT command that Joe can now execute are listed below:

```
GRANT INSERT, DELETE ON Reserves TO Yuppy WITH GRANT OPTION
GRANT SELECT ON Reserves TO Michael
GRANT SELECT ON Sailors TO Michael WITH Grant option
GRANT UPDATE (rating) ON Sailors TO Leah
```
Role-based authorization in SQL: Privileges are assigned to users (authorization ids, to be precise) in SQL-92. In the real world, privileges are often associated with a user’s job or role within the organization. Many DBMSs have long supported the concept of a role and allowed privileges to be assigned to roles. Roles can then be granted to users and other roles. (Of course, privileges can also be granted directly to users.) The SQL:1999 standard includes support for roles. What is the benefit of including a feature that many systems already support? This ensures that over time, all vendors who comply with the standard will support this feature. Thus, users can use the feature without worrying about portability of their application across DBMSs.

GRANT REFERENCES (bid) ON Boats TO Bill

Yuppy can insert or delete Reserves rows and can authorize someone else to do the same. Michael can execute SELECT queries on Sailors and Reserves, and he can pass this privilege to others for Sailors, but not for Reserves. With the SELECT privilege, Michael can create a view that accesses the Sailors and Reserves tables (for example, the ActiveSailors view) but he cannot grant SELECT on ActiveSailors to others.

On the other hand, suppose that Michael creates the following view:

CREATE VIEW YoungSailors (sid, age, rating)
AS SELECT S.sid, S.age, S.rating
FROM Sailors S
WHERE S.age < 18

The only underlying table is Sailors, for which Michael has SELECT with the grant option. He therefore has SELECT with the grant option on YoungSailors and can pass on the SELECT privilege on YoungSailors to Eric and Guppy:

GRANT SELECT ON YoungSailors TO Eric, Guppy

Eric and Guppy can now execute SELECT queries on the view YoungSailors—note, however, that Eric and Guppy do not have the right to execute SELECT queries directly on the underlying Sailors table.

Michael can also define constraints based on the information in the Sailors and Reserves tables. For example, Michael can define the following table, which has an associated table constraint:

CREATE TABLE Sneaky (maxrating INTEGER,
                      CHECK (maxrating >=
\[
\begin{align*}
&\text{( SELECT MAX (S.rating )} \\
&\text{ FROM Sailors S })})
\end{align*}
\]

By repeatedly inserting rows with gradually increasing \textit{maxrating} values into the Sneaky table until an insertion finally succeeds, Michael can find out the highest \textit{rating} value in the Sailors table! This example illustrates why SQL requires the creator of a table constraint that refers to Sailors to possess the \textbf{SELECT} privilege on Sailors.

Returning to the privileges granted by Joe, Leah can update only the \textit{rating} column of Sailors rows. She can execute the following command, which sets all ratings to 8:

\begin{verbatim}
UPDATE Sailors S \\
SET S.rating = 8
\end{verbatim}

However, she cannot execute the same command if the \textbf{SET} clause is changed to be \textbf{SET} \textit{S.age = 25}, because she is not allowed to update the \textit{age} field. A more subtle point is illustrated by the following command, which decrements the rating of all sailors:

\begin{verbatim}
UPDATE Sailors S \\
SET S.rating = S.rating - 1
\end{verbatim}

Leah cannot execute this command because it requires the \textbf{SELECT} privilege on the \textit{S.rating} column and Leah does not have this privilege!

Bill can refer to the \textit{bid} column of Boats as a foreign key in another table. For example, Bill can create the Reserves table through the following command:

\begin{verbatim}
CREATE TABLE Reserves ( sname CHAR(10) NOTNULL, \\
                      bid INTEGER, \\
                      day DATE, \\
                      PRIMARY KEY (bid, day), \\
                      UNIQUE (sname), \\
                      FOREIGN KEY (bid) REFERENCES Boats )
\end{verbatim}

If Bill did not have the \textbf{REFERENCES} privilege on the \textit{bid} column of Boats, he would not be able to execute this \textbf{CREATE} statement because the \textbf{FOREIGN KEY} clause requires this privilege.

Specifying just the \textbf{INSERT} (similarly, \textbf{REFERENCES} etc.) privilege in a \textbf{GRANT} command is not the same as specifying \textbf{SELECT(column-name)} for each column currently in the table. Consider the following command over the Sailors table, which has columns \textit{sid}, \textit{sname}, \textit{rating}, and \textit{age}:

\begin{verbatim}
GRANT INSERT ON Sailors TO Michael
\end{verbatim}
Suppose that this command is executed and then a column is added to the Sailors table (by executing an `ALTER TABLE` command). Note that Michael has the `INSERT` privilege with respect to the newly added column! If we had executed the following `GRANT` command, instead of the previous one, Michael would not have the `INSERT` privilege on the new column:

```
GRANT INSERT ON Sailors(sid), Sailors(sname), Sailors(rating),
            Sailors(age), TO Michael
```

There is a complementary command to `GRANT` that allows the withdrawal of privileges. The syntax of the `REVOKE` command is as follows:

```
REVOKE [ GRANT OPTION FOR ] privileges
          ON object FROM users { RESTRICT | CASCADE }
```

The command can be used to revoke either a privilege or just the grant option on a privilege (by using the optional `GRANT OPTION FOR` clause). One of the two alternatives, `RESTRICT` or `CASCADE`, must be specified; we will see what this choice means shortly.

The intuition behind the `GRANT` command is clear: The creator of a base table or a view is given all the appropriate privileges with respect to it and is allowed to pass these privileges—including the right to pass along a privilege!—to other users. The `REVOKE` command is, as expected, intended to achieve the reverse: A user who has granted a privilege to another user may change his mind and want to withdraw the granted privilege. The intuition behind exactly what effect a `REVOKE` command has is complicated by the fact that a user may be granted the same privilege multiple times, possibly by different users.

When a user executes a `REVOKE` command with the `CASCADE` keyword, the effect is to withdraw the named privileges or grant option from all users who currently hold these privileges solely through a `GRANT` command that was previously executed by the same user who is now executing the `REVOKE` command. If these users received the privileges with the grant option and passed it along, those recipients will also lose their privileges as a consequence of the `REVOKE` command unless they also received these privileges independently.

We illustrate the `REVOKE` command through several examples. First, consider what happens after the following sequence of commands, where Joe is the creator of Sailors.

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION    (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION     (executed by Art)
REVOKE SELECT ON Sailors FROM Art CASCADE           (executed by Joe)
```
Art loses the `SELECT` privilege on `Sailors`, of course. Then Bob, who received this privilege from Art, and only Art, also loses this privilege. Bob’s privilege is said to be **abandoned** when the privilege that it was derived from (Art’s `SELECT` privilege with grant option, in this example) is revoked. When the `CASCADE` keyword is specified, all abandoned privileges are also revoked (possibly causing privileges held by other users to become abandoned and thereby revoked recursively). If the `RESTRICT` keyword is specified in the `REVOKE` command, the command is rejected if revoking the privileges *just* from the users specified in the command would result in other privileges becoming abandoned.

Consider the following sequence, as another example:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION (executed by Art)
REVOKE SELECT ON Sailors FROM Art CASCADE (executed by Joe)
```

As before, Art loses the `SELECT` privilege on `Sailors`. But what about Bob? Bob received this privilege from Art, but he also received it independently (coincidentally, directly from Joe). Thus Bob retains this privilege. Consider a third example:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION (executed by Joe)
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION (executed by Joe)
REVOKE SELECT ON Sailors FROM Art CASCADE (executed by Joe)
```

Since Joe granted the privilege to Art twice and only revoked it once, does Art get to keep the privilege? As per the SQL-92 standard, no. Even if Joe absentmindedly granted the same privilege to Art several times, he can revoke it with a single `REVOKE` command.

It is possible to revoke just the grant option on a privilege:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION (executed by Joe)
REVOKE GRANT OPTION FOR SELECT ON Sailors FROM Art CASCADE (executed by Joe)
```

This command would leave Art with the `SELECT` privilege on `Sailors`, but Art no longer has the grant option on this privilege and therefore cannot pass it on to other users.

These examples bring out the intuition behind the `REVOKE` command, but they also highlight the complex interaction between `GRANT` and `REVOKE` commands. When a `GRANT` is executed, a **privilege descriptor** is added to a table of such descriptors maintained by the DBMS. The privilege descriptor specifies the following: the `grantor` of the privilege, the `grantee` who receives the privilege, the `granted privilege` (including
the name of the object involved), and whether the grant option is included. When a user creates a table or view and ‘automatically’ gets certain privileges, a privilege descriptor with system as the grantor is entered into this table.

The effect of a series of GRANT commands can be described in terms of an authorization graph in which the nodes are users—technically, they are authorization ids—and the arcs indicate how privileges are passed. There is an arc from (the node for) user 1 to user 2 if user 1 executed a GRANT command giving a privilege to user 2; the arc is labeled with the descriptor for the GRANT command. A GRANT command has no effect if the same privileges have already been granted to the same grantee by the same grantor. The following sequence of commands illustrates the semantics of GRANT and REVOKE commands when there is a cycle in the authorization graph:

```
GRANT SELECT ON Sailors TO Art WITH GRANT OPTION (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION (executed by Art)
GRANT SELECT ON Sailors TO Cal WITH GRANT OPTION (executed by Joe)
GRANT SELECT ON Sailors TO Bob WITH GRANT OPTION (executed by Cal)
REVOKE SELECT ON Sailors FROM Art CASCADE (executed by Joe)
```

The authorization graph for this example is shown in Figure 17.1. Note that we indicate how Joe, the creator of Sailors, acquired the SELECT privilege from the DBMS by introducing a System node and drawing an arc from this node to Joe’s node.

![Figure 17.1 Example Authorization Graph](image)

As the graph clearly indicates, Bob’s grant to Art and Art’s grant to Bob (of the same privilege) creates a cycle. Bob is subsequently given the same privilege by Cal, who received it independently from Joe. At this point Joe decides to revoke the privilege that he granted to Art.
Let us trace the effect of this revocation. The arc from Joe to Art is removed because it corresponds to the granting action that is revoked. All remaining nodes have the following property: If node $N$ has an outgoing arc labeled with a privilege, there is a path from the System node to node $N$ in which each arc label contains the same privilege plus the grant option. That is, any remaining granting action is justified by a privilege received (directly or indirectly) from the System. The execution of Joe’s `REVOKE` command therefore stops at this point, with everyone continuing to hold the `SELECT` privilege on Sailors.

This result may seem unintuitive because Art continues to have the privilege only because he received it from Bob, and at the time that Bob granted the privilege to Art, he had received it only from Art! Although Bob acquired the privilege through Cal subsequently, shouldn’t the effect of his grant to Art be undone when executing Joe’s `REVOKE` command? The effect of the grant from Bob to Art is not undone in SQL-92. In effect, if a user acquires a privilege multiple times from different grantors, SQL-92 treats each of these grants to the user as having occurred before that user passed on the privilege to other users. This implementation of `REVOKE` is convenient in many real-world situations. For example, if a manager is fired after passing on some privileges to subordinates (who may in turn have passed the privileges to others), we can ensure that only the manager’s privileges are removed by first redoing all of the manager’s granting actions and then revoking his or her privileges. That is, we need not recursively redo the subordinates’ granting actions.

To return to the saga of Joe and his friends, let us suppose that Joe decides to revoke Cal’s `SELECT` privilege as well. Clearly, the arc from Joe to Cal corresponding to the grant of this privilege is removed. The arc from Cal to Bob is removed as well, since there is no longer a path from System to Cal that gives Cal the right to pass the `SELECT` privilege on Sailors to Bob. The authorization graph at this intermediate point is shown in Figure 17.2.

The graph now contains two nodes (Art and Bob) for which there are outgoing arcs with labels containing the `SELECT` privilege on Sailors; thus, these users have granted this privilege. However, although each node contains an incoming arc carrying the same privilege, there is no such path from System to either of these nodes; thus, these users’ right to grant the privilege has been abandoned. We therefore remove the outgoing arcs as well. In general, these nodes might have other arcs incident upon them, but in this example, they now have no incident arcs. Joe is left as the only user with the `SELECT` privilege on Sailors; Art and Bob have lost their privileges.

### 17.3.1 Grant and Revoke on Views and Integrity Constraints *

The privileges held by the creator of a view (with respect to the view) change over time as he or she gains or loses privileges on the underlying tables. If the creator loses
a privilege held with the grant option, users who were given that privilege on the view will lose it as well. There are some subtle aspects to the `GRANT` and `REVOKE` commands when they involve views or integrity constraints. We will consider some examples that highlight the following important points:

1. A view may be dropped because a `SELECT` privilege is revoked from the user who created the view.

2. If the creator of a view gains additional privileges on the underlying tables, he or she automatically gains additional privileges on the view.

3. The distinction between the `REFERENCES` and `SELECT` privileges is important.

Suppose that Joe created Sailors and gave Michael the `SELECT` privilege on it with the grant option, and Michael then created the view YoungSailors and gave Eric the `SELECT` privilege on YoungSailors. Eric now defines a view called FineYoungSailors:

```sql
CREATE VIEW FineYoungSailors (name, age, rating)
AS SELECT S.sname, S.age, S.rating
FROM YoungSailors S
WHERE S.rating > 6
```

What happens if Joe revokes the `SELECT` privilege on Sailors from Michael? Michael no longer has the authority to execute the query used to define YoungSailors because the definition refers to Sailors. Therefore, the view YoungSailors is dropped (i.e., destroyed). In turn, FineYoungSailors is dropped as well. Both these view definitions are removed from the system catalogs; even if a remorseful Joe decides to give back
the `SELECT` privilege on Sailors to Michael, the views are gone and must be created afresh if they are required.

On a more happy note, suppose that everything proceeds as described above until Eric defines FineYoungSailors; then, instead of revoking the `SELECT` privilege on Sailors from Michael, Joe decides to also give Michael the `INSERT` privilege on Sailors. Michael’s privileges on the view YoungSailors are upgraded to what he would have if he were to create the view `now`. Thus he acquires the `INSERT` privilege on YoungSailors as well. (Note that this view is updatable.) What about Eric? His privileges are unchanged.

Whether or not Michael has the `INSERT` privilege on YoungSailors with the grant option depends on whether or not Joe gives him the `INSERT` privilege on Sailors with the grant option. To understand this situation, consider Eric again. If Michael has the `INSERT` privilege on YoungSailors with the grant option, he can pass this privilege to Eric. Eric could then insert rows into the Sailors table because inserts on YoungSailors are effected by modifying the underlying base table, Sailors. Clearly, we don’t want Michael to be able to authorize Eric to make such changes unless Michael has the `INSERT` privilege on Sailors with the grant option.

The `REFERENCES` privilege is very different from the `SELECT` privilege, as the following example illustrates. Suppose that Joe is the creator of Boats. He can authorize another user, say Fred, to create Reserves with a foreign key that refers to the `bid` column of Boats by giving Fred the `REFERENCES` privilege with respect to this column. On the other hand, if Fred has the `SELECT` privilege on the `bid` column of Boats but not the `REFERENCES` privilege, Fred cannot create Reserves with a foreign key that refers to Boats. If Fred creates Reserves with a foreign key column that refers to `bid` in Boats, and later loses the `REFERENCES` privilege on the `bid` column of boats, the foreign key constraint in Reserves is dropped; however, the Reserves table is not dropped.

To understand why the SQL-92 standard chose to introduce the `REFERENCES` privilege, rather than to simply allow the `SELECT` privilege to be used in this situation, consider what happens if the definition of Reserves specified the `NO ACTION` option with the foreign key—Joe, the owner of Boats, may be prevented from deleting a row from Boats because a row in Reserves refers to this Boats row! Giving Fred, the creator of Reserves, the right to constrain updates on Boats in this manner goes beyond simply allowing him to read the values in Boats, which is all that the `SELECT` privilege authorizes.

### 17.4 MANDATORY ACCESS CONTROL *

Discretionary access control mechanisms, while generally effective, have certain weaknesses. In particular they are susceptible to Trojan horse schemes whereby a devious
Unauthorized user can trick an authorized user into disclosing sensitive data. For example, suppose that student Tricky Dick wants to break into the grade tables of instructor Trustin Justin. Dick does the following:

- He creates a new table called MineAllMine and gives `INSERT` privileges on this table to Justin (who is blissfully unaware of all this attention, of course).
- He modifies the code of some DBMS application that Justin uses often to do a couple of additional things: first, read the Grades table, and next, write the result into MineAllMine.

Then he sits back and waits for the grades to be copied into MineAllMine and later undoes the modifications to the application to ensure that Justin does not somehow find out later that he has been cheated. Thus, despite the DBMS enforcing all discretionary access controls—only Justin’s authorized code was allowed to access Grades—sensitive data is disclosed to an intruder. The fact that Dick could surreptitiously modify Justin’s code is outside the scope of the DBMS’s access control mechanism.

Mandatory access control mechanisms are aimed at addressing such loopholes in discretionary access control. The popular model for mandatory access control, called the Bell-LaPadula model, is described in terms of objects (e.g., tables, views, rows, columns), subjects (e.g., users, programs), security classes, and clearances. Each database object is assigned a security class, and each subject is assigned clearance for a security class; we will denote the class of an object or subject $A$ as $\text{class}(A)$. The security classes in a system are organized according to a partial order, with a most secure class and a least secure class. For simplicity, we will assume that there are four classes: top secret (TS), secret (S), confidential (C), and unclassified (U). In this system, $\text{TS} > \text{S} > \text{C} > \text{U}$, where $A > B$ means that class $A$ data is more sensitive than class $B$ data.

The Bell-LaPadula model imposes two restrictions on all reads and writes of database objects:

1. **Simple Security Property:** Subject $S$ is allowed to read object $O$ only if $\text{class}(S) \geq \text{class}(O)$. For example, a user with TS clearance can read a table with C clearance, but a user with C clearance is not allowed to read a table with TS classification.

2. ***-Property:** Subject $S$ is allowed to write object $O$ only if $\text{class}(S) \leq \text{class}(O)$. For example, a user with S clearance can only write objects with S or TS classification.

If discretionary access controls are also specified, these rules represent additional restrictions. Thus, to read or write a database object, a user must have the necessary privileges (obtained via GRANT commands) and the security classes of the user and the object must satisfy the preceding restrictions. Let us consider how such a mandatory
control mechanism might have foiled Tricky Dick. The Grades table could be classified as $S$, Justin could be given clearance for $S$, and Tricky Dick could be given a lower clearance ($C$). Dick can only create objects of $C$ or lower classification; thus, the table MineAllMine can have at most the classification $C$. When the application program running on behalf of Justin (and therefore with clearance $S$) tries to copy Grades into MineAllMine, it is not allowed to do so because $\text{class}(\text{MineAllMine}) < \text{class(application)}$, and the *-Property is violated.

### 17.4.1 Multilevel Relations and Polyinstantiation

To apply mandatory access control policies in a relational DBMS, a security class must be assigned to each database object. The objects can be at the granularity of tables, rows, or even individual column values. Let us assume that each row is assigned a security class. This situation leads to the concept of a **multilevel table**, which is a table with the surprising property that users with different security clearances will see a different collection of rows when they access the same table.

Consider the instance of the Boats table shown in Figure 17.3. Users with $S$ and $TS$ clearance will get both rows in the answer when they ask to see all rows in Boats. A user with $C$ clearance will get only the second row, and a user with $U$ clearance will get no rows.

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
<th>Security Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Salsa</td>
<td>Red</td>
<td>$S$</td>
</tr>
<tr>
<td>102</td>
<td>Pinto</td>
<td>Brown</td>
<td>$C$</td>
</tr>
</tbody>
</table>

**Figure 17.3** An Instance $B1$ of Boats

The Boats table is defined to have $bid$ as the primary key. Suppose that a user with clearance $C$ wishes to enter the row $\langle 101, \text{Picante}, \text{Scarlet}, C \rangle$. We have a dilemma:

- If the insertion is permitted, two distinct rows in the table will have key 101.
- If the insertion is not permitted because the primary key constraint is violated, the user trying to insert the new row, who has clearance $C$, can infer that there is a boat with $bid=101$ whose security class is higher than $C$. This situation compromises the principle that users should not be able to infer any information about objects that have a higher security classification.

This dilemma is resolved by effectively treating the security classification as part of the key. Thus, the insertion is allowed to continue, and the table instance is modified as shown in Figure 17.4.
Users with clearance $C$ or $U$ see just the rows for Picante and Pinto, but users with clearance $S$ or $TS$ see all three rows. The two rows with $bid=101$ can be interpreted in one of two ways: only the row with the higher classification (Salsa, with classification $S$) actually exists, or both exist and their presence is revealed to users according to their clearance level. The choice of interpretation is up to application developers and users.

The presence of data objects that appear to have different values to users with different clearances (for example, the boat with $bid=101$) is called polyinstantiation. If we consider security classifications associated with individual columns, the intuition underlying polyinstantiation can be generalized in a straightforward manner, but some additional details must be addressed. We remark that the main drawback of mandatory access control schemes is their rigidity; policies are set by system administrators, and the classification mechanisms are not flexible enough. A satisfactory combination of discretionary and mandatory access controls is yet to be achieved.

### 17.4.2 Covert Channels, DoD Security Levels

Even if a DBMS enforces the mandatory access control scheme discussed above, information can flow from a higher classification level to a lower classification level through indirect means, called covert channels. For example, if a transaction accesses data at more than one site in a distributed DBMS, the actions at the two sites must be coordinated. The process at one site may have a lower clearance (say $C$) than the process at another site (say $S$), and both processes have to agree to commit before the transaction can be committed. This requirement can be exploited to pass information with an $S$ classification to the process with a $C$ clearance: The transaction is repeatedly invoked, and the process with the $C$ clearance always agrees to commit, whereas the process with the $S$ clearance agrees to commit if it wants to transmit a 1 bit and does not agree if it wants to transmit a 0 bit.

In this (admittedly tortuous) manner, information with an $S$ clearance can be sent to a process with a $C$ clearance as a stream of bits. This covert channel is an indirect violation of the intent behind the *-Property. Additional examples of covert channels can be found readily in statistical databases, which we discuss in Section 17.5.2.
Current systems: Commercial RDBMSs are available that support discretionary controls at the $C_2$ level and mandatory controls at the $B_1$ level. IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all support SQL-92’s features for discretionary access control. In general, they do not support mandatory access control; Oracle does offer a version of their product with support for mandatory access control.

DBMS vendors have recently started implementing mandatory access control mechanisms (although they are not part of the SQL-92 standard) because the United States Department of Defense (DoD) requires such support for its systems. The DoD requirements can be described in terms of security levels $A$, $B$, $C$, and $D$ of which $A$ is the most secure and $D$ is the least secure.

Level $C$ requires support for discretionary access control. It is divided into sublevels $C_1$ and $C_2$; $C_2$ also requires some degree of accountability through procedures such as login verification and audit trails. Level $B$ requires support for mandatory access control. It is subdivided into levels $B_1$, $B_2$, and $B_3$. Level $B_2$ additionally requires the identification and elimination of covert channels. Level $B_3$ additionally requires maintenance of audit trails and the designation of a security administrator (usually, but not necessarily, the DBA). Level $A$, the most secure level, requires a mathematical proof that the security mechanism enforces the security policy!

17.5 ADDITIONAL ISSUES RELATED TO SECURITY *

Security is a broad topic, and our coverage is necessarily limited. This section briefly touches on some additional important issues.

17.5.1 Role of the Database Administrator

The database administrator (DBA) plays an important role in enforcing the security-related aspects of a database design. In conjunction with the owners of the data, the DBA will probably also contribute to developing a security policy. The DBA has a special account, which we will call the system account, and is responsible for the overall security of the system. In particular the DBA deals with the following:

1. Creating new accounts: Each new user or group of users must be assigned an authorization id and a password. Note that application programs that access the database have the same authorization id as the user executing the program.

2. Mandatory control issues: If the DBMS supports mandatory control—some customized systems for applications with very high security requirements (for
example, military data) provide such support—the DBA must assign security
classes to each database object and assign security clearances to each authorization
id in accordance with the chosen security policy.

The DBA is also responsible for maintaining the audit trail, which is essentially the
log of updates with the authorization id (of the user who is executing the transaction)
added to each log entry. This log is just a minor extension of the log mechanism
used to recover from crashes. Additionally, the DBA may choose to maintain a log
of all actions, including reads, performed by a user. Analyzing such histories of how
the DBMS was accessed can help prevent security violations by identifying suspicious
patterns before an intruder finally succeeds in breaking in, or it can help track down
an intruder after a violation has been detected.

17.5.2 Security in Statistical Databases

A statistical database is one that contains specific information on individuals or
events but is intended to permit only statistical queries. For example, if we maintained
a statistical database of information about sailors, we would allow statistical queries
about average ratings, maximum age, and so on, but would not want to allow queries
about individual sailors. Security in such databases poses some new problems because
it is possible to infer protected information (such as an individual sailor’s rating) from
answers to permitted statistical queries. Such inference opportunities represent covert
channels that can compromise the security policy of the database.

Suppose that sailor Sneaky Pete wants to know the rating of Admiral Horntooter, the
esteemed chairman of the sailing club, and happens to know that Horntooter is the
oldest sailor in the club. Pete repeatedly asks queries of the form “How many sailors
are there whose age is greater than \(X\)?” for various values of \(X\), until the answer is 1.
Obviously, this sailor is Horntooter, the oldest sailor. Note that each of these queries
is a valid statistical query and is permitted. Let the value of \(X\) at this point be, say,
65. Pete now asks the query, “What is the maximum rating of all sailors whose age
is greater than 65?” Again, this query is permitted because it is a statistical query.
However, the answer to this query reveals Horntooter’s rating to Pete, and the security
policy of the database is violated.

One approach to preventing such violations is to require that each query must involve
at least some minimum number, say \(N\), of rows. With a reasonable choice of \(N\), Pete
would not be able to isolate the information about Horntooter, because the query
about the maximum rating would fail. This restriction, however, is easy to overcome.
By repeatedly asking queries of the form, “How many sailors are there whose age is
greater than \(X\)?” until the system rejects one such query, Pete identifies a set of \(N\)
sailors, including Horntooter. Let the value of \(X\) at this point be 55. Now, Pete can
ask two queries:
Chapter 17

- “What is the sum of the ratings of all sailors whose age is greater than 55?” Since $N$ sailors have age greater than 55, this query is permitted.
- “What is the sum of the ratings of all sailors, other than Horntooter, whose age is greater than 55, and sailor Pete?” Since the set of sailors whose ratings are added up now includes Pete instead of Horntooter, but is otherwise the same, the number of sailors involved is still $N$, and this query is also permitted.

From the answers to these two queries, say $A_1$ and $A_2$, Pete, who knows his rating, can easily calculate Horntooter’s rating as $A_1 - A_2 + $Pete’s rating$.$

Pete succeeded because he was able to ask two queries that involved many of the same sailors. The number of rows examined in common by two queries is called their intersection. If a limit were to be placed on the amount of intersection permitted between any two queries issued by the same user, Pete could be foiled. Actually, a truly fiendish (and patient) user can generally find out information about specific individuals even if the system places a minimum number of rows bound ($N$) and a maximum intersection bound ($M$) on queries, but the number of queries required to do this grows in proportion to $N/M$. We can try to additionally limit the total number of queries that a user is allowed to ask, but two users could still conspire to breach security. By maintaining a log of all activity (including read-only accesses), such query patterns can be detected, hopefully before a security violation occurs. This discussion should make it clear, however, that security in statistical databases is difficult to enforce.

17.5.3 Encryption

A DBMS can use encryption to protect information in certain situations where the normal security mechanisms of the DBMS are not adequate. For example, an intruder may steal tapes containing some data or tap a communication line. By storing and transmitting data in an encrypted form, the DBMS ensures that such stolen data is not intelligible to the intruder.

The basic idea behind encryption is to apply an encryption algorithm, which may be accessible to the intruder, to the original data and a user-specified or DBA-specified encryption key, which is kept secret. The output of the algorithm is the encrypted version of the data. There is also a decryption algorithm, which takes the encrypted data and the encryption key as input and then returns the original data. Without the correct encryption key, the decryption algorithm produces gibberish. This approach forms the basis for the Data Encryption Standard (DES), which has been in use since 1977, with an encryption algorithm that consists of character substitutions and permutations. The main weakness of this approach is that authorized users must be told the encryption key, and the mechanism for communicating this information is vulnerable to clever intruders.
Another approach to encryption, called public-key encryption, has become increasingly popular in recent years. The encryption scheme proposed by Rivest, Shamir, and Adleman, called RSA, is a well-known example of public-key encryption. Each authorized user has a public encryption key, known to everyone, and a private decryption key (used by the decryption algorithm), chosen by the user and known only to him or her. The encryption and decryption algorithms themselves are assumed to be publicly known. Consider a user called Sam. Anyone can send Sam a secret message by encrypting the message using Sam’s publicly known encryption key. Only Sam can decrypt this secret message because the decryption algorithm requires Sam’s decryption key, known only to Sam. Since users choose their own decryption keys, the weakness of DES is avoided.

The main issue for public-key encryption is how encryption and decryption keys are chosen. Technically, public-key encryption algorithms rely on the existence of one-way functions, which are functions whose inverse is computationally very hard to determine. The RSA algorithm, for example, is based on the observation that although checking whether a given number is prime is easy, determining the prime factors of a nonprime number is extremely hard. (Determining the prime factors of a number with over 100 digits can take years of CPU-time on the fastest available computers today.)

We now sketch the intuition behind the RSA algorithm, assuming that the data to be encrypted is an integer $I$. To choose an encryption key and a decryption key, our friend Sam would first choose a very large integer $limit$, which we assume is larger than the largest integer that he will ever need to encode. Sam chooses $limit$ to be the product of two (large!) distinct prime numbers, say $p \times q$. Sam then chooses some prime number $e$, chosen to be larger than both $p$ and $q$, as his encryption key. Both $limit$ and $e$ are made public and are used by the encryption algorithm.

Now comes the clever part: Sam chooses the decryption key $d$ in a special way based on $p$, $q$, and $e$. The essential point of the scheme is that it is easy to compute $d$ given $e$, $p$, and $q$, but very hard to compute $d$ given just $e$ and $limit$. In turn, this difficulty depends on the fact that it is hard to determine the prime factors of $limit$, which happen to be $p$ and $q$.

A very important property of the encryption and decryption algorithms in this scheme is that given the corresponding encryption and decryption keys, the algorithms are inverses of each other—not only can data be encrypted and then decrypted, but we can also apply the decryption algorithm first and then the encryption algorithm and still get the original data back! This property can be exploited by two users, say Elmer and Sam, to exchange messages in such a way that if Elmer gets a message that is supposedly from Sam, he can verify that it is from Sam (in addition to being able to decrypt the message), and further, prove that it is from Sam. This feature has obvious

\[1\text{In case you are curious, } d \text{ is chosen such that } d \times e = 1 \mod ((p-1) \times (q-1)).\]
practical value. For example, suppose that Elmer’s company accepts orders for its products over the Internet and stores these orders in a DBMS. The requirements are:

1. Only the company (Elmer) should be able to understand an order. A customer (say Sam) who orders jewelry frequently may want to keep the orders private (perhaps because he does not want to become a popular attraction for burglars!).

2. The company should be able to verify that an order that supposedly was placed by customer Sam was indeed placed by Sam, and not by an intruder claiming to be Sam. By the same token, Sam should not be able to claim that the company forged an order from him—an order from Sam must provably come from Sam.

The company asks each customer to choose an encryption key (Sam chooses $e_{Sam}$) and a decryption key ($d_{Sam}$) and to make the encryption key public. It also makes its own encryption key ($e_{Elmer}$) public. The company’s decryption key ($d_{Elmer}$) is kept secret, and customers are expected to keep their decryption keys secret as well.

Now let’s see how the two requirements can be met. To place an order, Sam could just encrypt the order using encryption key $e_{Elmer}$, and Elmer could decrypt this using decryption key $d_{Elmer}$. This simple approach satisfies the first requirement because $d_{Elmer}$ is known only to Elmer. However, since $e_{Elmer}$ is known to everyone, someone who wishes to play a prank could easily place an order on behalf of Sam without informing Sam. From the order itself, there is no way for Elmer to verify that it came from Sam. (Of course, one way to handle this is to give each customer an account and to rely on the login procedure to verify the identity of the user placing the order—the user would have to know the password for Sam’s account—but the company may have thousands of customers and may not want to give each of them an account.)

A clever use of the encryption scheme, however, allows Elmer to verify whether the order was indeed placed by Sam. Instead of encrypting the order using $e_{Elmer}$, Sam first applies his decryption algorithm, using $d_{Sam}$, known only to Sam (and not even to Elmer!), to the original order. Since the order was not encrypted first, this produces gibberish, but as we shall see, there is a method in this madness. Next, Sam encrypts the result of the previous step using $e_{Elmer}$ and registers the result in the database.

When Elmer examines such an order, he first decrypts it using $d_{Elmer}$. This step yields the gibberish that Sam generated from his order, because the encryption and decryption algorithm are inverses when applied with the right keys. Next, Elmer applies the encryption algorithm to this gibberish, using Sam’s encryption key $e_{Sam}$, which is known to Elmer (and is public). This step yields the original unencrypted order, again because the encryption and decryption algorithm are inverses!

If the order had been forged, the forger could not have known Sam’s decryption key $d_{Sam}$; the final result would have been nonsensical, rather than the original order.
Further, because the company does not know $d_{Sam}$, Sam cannot claim that a genuine order was forged by the company.

The use of public-key cryptography is not limited to database systems, but it is likely to find increasing application in the DBMS context thanks to the use of the DBMS as a repository for the records of sensitive commercial transactions. Internet commerce, as in the example above, could be a driving force in this respect.

### 17.6 POINTS TO REVIEW

- There are three main security objectives. First, information should not be disclosed to unauthorized users (secrecy). Second, only authorized users should be allowed to modify data (integrity). Third, authorized users should not be denied access (availability). A security policy describes the security measures enforced. These measures use the security mechanisms of the underlying DBMS. (Section 17.1)

- There are two main approaches to enforcing security measures. In discretionary access control, users have privileges to access or modify objects in the database. If they have permission, users can grant their privileges to other users, and the DBMS keeps track of who has what rights. In mandatory access control, objects are assigned security classes. Users have security clearance for a security class. Rules involving the security class and a user’s clearance determine which database objects the user can access. (Section 17.2)

- SQL supports discretionary access through the GRANT and REVOKE commands. The creator of a table has automatically all privileges on it and can pass privileges on to other users or revoke privileges from other users. The effect of GRANT commands can be described as adding edges into an authorization graph and the effect of REVOKE commands can be described as removing edges from the graph. (Section 17.3)

- In mandatory access control, objects are organized into several security classes and users are organized into several levels of clearance. The security classes form a partial order. Reads and writes of an object are restricted by rules that involve the security class of the object and the clearance of the user. Users with different levels of clearance might see different records in the same table. This phenomenon is called polyinstantiation. (Section 17.4)

- The database administrator is responsible for the overall security of the system. The DBA has a system account with special privileges. The DBA also maintains an audit trail, a log of accesses to the DBMS with the corresponding user identifiers. Statistical databases only allow summary queries, but clever users can infer information about specific individuals from the answers to valid statistical queries.
We can use *encryption* techniques to ensure that stolen data cannot be deciphered.  
*(Section 17.5)*

**EXERCISES**

**Exercise 17.1** Briefly answer the following questions based on this schema:

```sql
Emp(eid: integer, ename: string, age: integer, salary: real)
Works(eid: integer, did: integer, pct_time: integer)
Dept(did: integer, budget: real, managerid: integer)
```

1. Suppose you have a view SeniorEmp defined as follows:
   ```sql
   CREATE VIEW SeniorEmp (sname, sage, salary) 
   AS SELECT E.ename, E.age, E.salary 
   FROM Emp E 
   WHERE E.age > 50
   ```

   Explain what the system will do to process the following query:
   ```sql
   SELECT S.sname 
   FROM SeniorEmp S 
   WHERE S.salary > 100,000
   ```

2. Give an example of a view on Emp that could be automatically updated by updating Emp.

3. Give an example of a view on Emp that would be impossible to update (automatically) and explain why your example presents the update problem that it does.

4. Consider the following view definition:
   ```sql
   CREATE VIEW DInfo (did, manager, numemps, totsals) 
   AS SELECT D.did, D.managerid, COUNT (*), SUM (E.salary) 
   FROM Emp E, Works W, Dept D 
   WHERE E.eid = W.eid AND W.did = D.did 
   GROUP BY D.did, D.managerid
   ```

   (a) Give an example of a view update on DInfo that could (in principle) be implemented automatically by updating one or more of the relations Emp, Works, and Dept. Does SQL-92 allow such a view update?

   (b) Give an example of a view update on DInfo that cannot (even in principle) be implemented automatically by updating one or more of the relations Emp, Works, and Dept. Explain why.

   (c) How could the view DInfo help in enforcing security?

**Exercise 17.2** You are the DBA for the VeryFine Toy Company, and you create a relation called Employees with fields ename, dept, and salary. For authorization reasons, you also define views EmployeeNames (with ename as the only attribute) and DeptInfo with fields dept and avgsalary. The latter lists the average salary for each department.
1. Show the view definition statements for EmployeeNames and DeptInfo.

2. What privileges should be granted to a user who needs to know only average department salaries for the Toy and CS departments?

3. You want to authorize your secretary to fire people (you'll probably tell him whom to fire, but you want to be able to delegate this task), to check on who is an employee, and to check on average department salaries. What privileges should you grant?

4. Continuing with the preceding scenario, you don't want your secretary to be able to look at the salaries of individuals. Does your answer to the previous question ensure this? Be specific: Can your secretary possibly find out salaries of some individuals (depending on the actual set of tuples), or can your secretary always find out the salary of any individual that he wants to?

5. You want to give your secretary the authority to allow other people to read the EmployeeNames view. Show the appropriate command.

6. Your secretary defines two new views using the EmployeeNames view. The first is called AtoRNames and simply selects names that begin with a letter in the range A to R. The second is called HowManyNames and counts the number of names. You are so pleased with this achievement that you decide to give your secretary the right to insert tuples into the EmployeeNames view. Show the appropriate command, and describe what privileges your secretary has after this command is executed.

7. Your secretary allows Todd to read the EmployeeNames relation and later quits. You then revoke the secretary’s privileges. What happens to Todd’s privileges?

8. Give an example of a view update on the above schema that cannot be implemented through updates to Employees.

9. You decide to go on an extended vacation, and to make sure that emergencies can be handled, you want to authorize your boss Joe to read and modify the Employees relation and the EmployeeNames relation (and Joe must be able to delegate authority, of course, since he’s too far up the management hierarchy to actually do any work). Show the appropriate SQL statements. Can Joe read the DeptInfo view?

10. After returning from your (wonderful) vacation, you see a note from Joe, indicating that he authorized his secretary Mike to read the Employees relation. You want to revoke Mike’s SELECT privilege on Employees, but you don’t want to revoke the rights that you gave to Joe, even temporarily. Can you do this in SQL?

11. Later you realize that Joe has been quite busy. He has defined a view called AllNames using the view EmployeeNames, defined another relation called StaffNames that he has access to (but that you can’t access), and given his secretary Mike the right to read from the AllNames view. Mike has passed this right on to his friend Susan. You decide that even at the cost of annoying Joe by revoking some of his privileges, you simply have to take away Mike and Susan’s rights to see your data. What REVOKE statement would you execute? What rights does Joe have on Employees after this statement is executed? What views are dropped as a consequence?

Exercise 17.3 Briefly answer the following questions.

1. Explain the intuition behind the two rules in the Bell-LaPadula model for mandatory access control.
2. Give an example of how covert channels can be used to defeat the Bell-LaPadula model.

3. Give an example of polyinstantiation.

4. Describe a scenario in which mandatory access controls prevent a breach of security that cannot be prevented through discretionary controls.

5. Describe a scenario in which discretionary access controls are required to enforce a security policy that cannot be enforced using only mandatory controls.

6. If a DBMS already supports discretionary and mandatory access controls, is there a need for encryption?

7. Explain the need for each of the following limits in a statistical database system:
   a. A maximum on the number of queries a user can pose.
   b. A minimum on the number of tuples involved in answering a query.
   c. A maximum on the intersection of two queries (i.e., on the number of tuples that both queries examine).

8. Explain the use of an audit trail, with special reference to a statistical database system.

9. What is the role of the DBA with respect to security?

10. What is public-key encryption? How does it differ from the encryption approach taken in the Data Encryption Standard (DES), and in what ways is it better than DES?

11. What are one-way functions, and what role do they play in public-key encryption?

12. Explain how a company offering services on the Internet could use public-key encryption to make its order-entry process secure. Describe how you would use DES encryption for the same purpose, and contrast the public-key and DES approaches.

PROJECT-BASED EXERCISES

Exercise 17.4 Is there any support for views or authorization in Minibase?

BIBLIOGRAPHIC NOTES

The authorization mechanism of System R, which greatly influenced the GRANT and REVOKE paradigm in SQL-92, is described in [290]. A good general treatment of security and cryptography is presented in [179], and an overview of database security can be found in [119] and [404]. Security in statistical databases is investigated in several papers, including [178] and [148]. Multilevel security is discussed in several papers, including [348, 434, 605, 621].