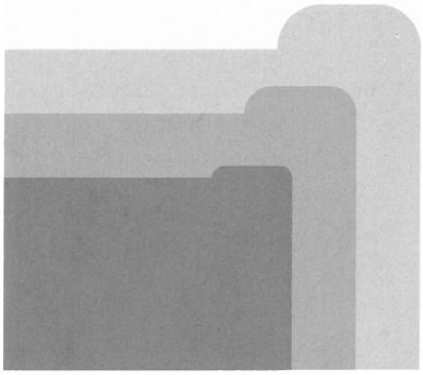
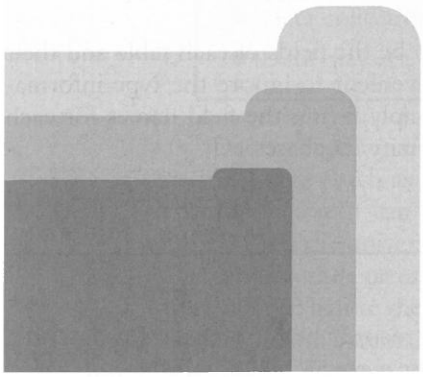
PART 1
Relational Databases



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

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

2
Data Definition



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

2.1 Tables

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

Each field of a table has a specified *type* and all record values for that field must be of that type. The database of Figure 1-1 is fairly simple, in that all of the fields either have the type *integer* or *character string*. Commercial database systems typically support

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

Figure 2-1
The schema of the university database

many types, including various numeric, string, and date/time types; these types are discussed in Section 4.3.2.†

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

2.2 Null Values

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

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

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

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

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

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

Null values occur for two reasons:

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

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

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

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

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

2.3 Keys

In the relational model, a user cannot reference a record by specifying its position (as in "I want the second record in STUDENT"). Instead, a user must reference a record by specifying field values (as in "I want the record for the student named Joe who

graduated in 1997"). But not all field values are guaranteed to uniquely identify a record. For example, specifying the student name and graduation year (as in the above request) is not sufficient, because it is possible for two students to have the same name and graduation year. On the other hand, supplying the student ID would be sufficient, because every student has a different ID.

A unique identifier is called a *superkey*.

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

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

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

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

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

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

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

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

same section twice. If we consider the table SECTION, each of the following assumptions results in a different key:

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

Most relational database systems do not require a table to have a key. However, the absence of a key means that the table could contain duplicate records (see Exercise 2.4). Duplicate records can cause problems, because they cannot be distinguished from each other. (Remember, we can access a record only by its values, not by its position.) For example, suppose that there are two students having the same name, major, and graduation year. If we remove the key field *SId* from the STUDENT table, then the records for these two students are duplicates. There now is no way to treat these students independently. In particular, we cannot change the major of one without changing the major of the other, nor can we enroll them in different courses or give them different grades.

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

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

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

2.4 Foreign Keys and Referential Integrity

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

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

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

Figure 2-2
Foreign keys for the university database

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

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

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

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

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

Specifying a foreign key has two benefits:

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

Consider the field *MajorId*. If this field is not specified as a foreign key, then a user is forced to guess at its meaning. By specifying the field as a foreign key, the creator documents its purpose—namely, as a reference to a particular DEPT record. This documentation helps

users understand how the tables are connected to each other, and tells them when it is appropriate to connect values between tables.

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

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

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

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

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

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

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

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

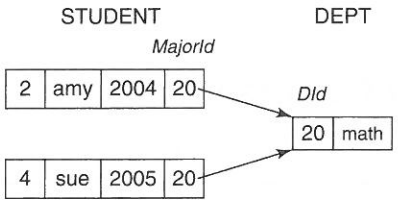


Figure 2-3
An example of referential integrity

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

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

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

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

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

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

2.5 Integrity Constraints

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

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

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

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

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

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

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

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

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

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

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

The decision to specify an integrity constraint is a two-edged sword. On one hand, the enforcement of a constraint helps to weed out bad data. On the other hand, it takes time (and possibly a lot of time) to enforce each constraint. As a practical matter, the database designer must weigh the time required to enforce a constraint against its

expected benefits. For example, computer science majors did not exist prior to 1960, so a valid constraint is that students graduating before 1960 should not have "compsci" as their major. Should we specify this constraint in the database?

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

2.6 Specifying Tables in SQL

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

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

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

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

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

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

Figure 2-4
The SQL specification of the STUDENT table

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

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

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

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

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

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

2.7 Chapter Summary

- The data in a relational database is organized into *tables*. Each table contains zero or more *records* and one or more *fields*.
- Each field has a specified *type*. Commercial database systems support many types, including various numeric, string, and date/time types.

- A *constraint* restricts the allowable records in a table. The database system ensures that the constraints are satisfied by refusing to execute any update request that would violate them.
- There are four important kinds of constraint: *integrity constraints, null value constraints, key constraints,* and *referential integrity constraints.* Constraints need to be specified explicitly; they cannot be inferred from the current contents of a table.
- An *integrity constraint* encodes "business rules" about the organization. An integrity constraint may apply to an individual record ("a student's graduation year is at least 1863"), a table ("a professor teaches at most two sections per year"), or database ("a student cannot take a course more than once").
- A *null value* is a placeholder for a missing or unknown value. A *null value constraint* asserts that a record cannot have a null value for the specified field. Null values have a complex semantics and should be disallowed for as many fields as possible. However, null values cannot always be avoided, because data collection may be incomplete and data may arrive late.
- A *key* is a minimal set of fields that can be used to identify a record. Specifying a key constrains the table so that no two records can have the same values for those fields. Although a table can have several keys, one key is chosen to be the *primary key.* Primary key fields must never be null.
- A *foreign key* is a set of fields from one table that corresponds to the primary key of another table. A foreign key value in one record uniquely identifies a record in the other table, thereby connecting the two records. The specification of a foreign key asserts *referential integrity,* which requires each non-null foreign key value to be the key value of some record.
- A table is specified in SQL using the *create table* command. This command specifies the name of the table, the name and type of its fields, and any constraints.
  - An integrity constraint on individual records is specified by means of the *check* keyword, followed by the constraint expression.
  - A null value constraint is specified by adding the phrase "not null" to a field specification.
  - A primary key is specified by means of the *key* keyword, followed by placing the key fields within parentheses, separated by commas.
  - A foreign key is specified by means of the *foreign key* keyword, followed by placing the foreign key fields within parentheses and specifying the table that the foreign key refers to.
- A foreign key constraint can also specify the action that the database system is to perform when an update violates referential integrity. The action can be one of *cascade*, *set null*, *set default*, and *no action.*

2.8 Suggested Reading

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

2.9 Exercises

CONCEPTUAL EXERCISES

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

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

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

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

2.5 Figure 2-2 asserts that *MajorId* is a foreign key of DEPT.

a) Explain why deleting a record from STUDENT does not violate referential integrity.

b) Explain why inserting a record into DEPT does not violate referential integrity.

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

2.7 An online bookseller stores information in the following tables:

```
BOOK(BookId, Title, AuthorName, Price)
CUSTOMER(CustId, CustName, Address)
CART_ITEM(CustId, BookId)
PURCHASE(PId, PurchaseDate, CustId)
PURCHASED_ITEM(PId, BookId)
```

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

a) What should we do about keys for PURCHASED_ITEM and CART_ITEM?

b) Give the foreign keys of each table.

c) List in English some reasonable integrity constraints that these tables might have.

d) Specify these tables in SQL.

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

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