# 3 THE RELATIONAL MODEL

> TABLE: An arrangement of words, numbers, or signs, or combinations of them, as in parallel columns, to exhibit a set of facts or relations in a definite, compact, and comprehensive form; a synopsis or scheme.
>
> —Webster's *Dictionary of the English Language*

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**.

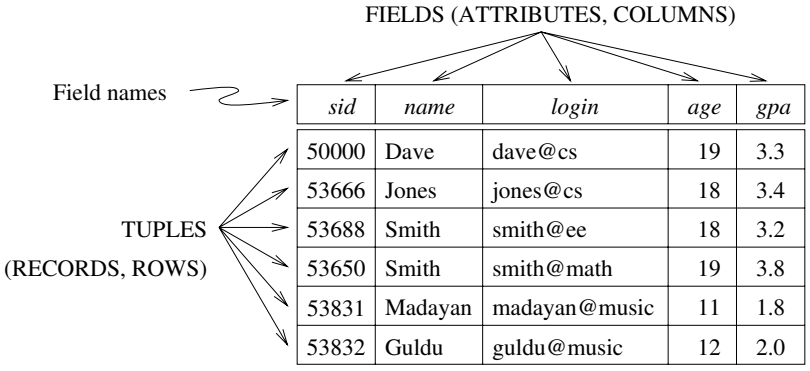
We use the example of student information in a university database from Chapter 1 to illustrate the parts of a relation schema:

Students(*sid:* `string`, *name:* `string`, *login:* `string`, *age:* `integer`, *gpa:* `real`)

This says, for instance, that the field named *sid* has a domain named `string`. The set of values associated with domain `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 *relation instance* is often abbreviated to just *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 $S1$ contains




| *sid* | *name* | *login* | *age* | *gpa* |
|---|---|---|---|---|
| 50000 | Dave | dave@cs | 19 | 3.3 |
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@ee | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |
| 53831 | Madayan | madayan@music | 11 | 1.8 |
| 53832 | Guldu | guldu@music | 12 | 2.0 |

**Figure 3.1** An Instance $S1$ of the Students Relation

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.

| *sid* | *name* | *login* | *age* | *gpa* |
|---|---|---|---|---|
| 53831 | Madayan | madayan@music | 11 | 1.8 |
| 53832 | Guldu | guldu@music | 12 | 2.0 |
| 53688 | Smith | smith@ee | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 50000 | Dave | dave@cs | 19 | 3.3 |

**Figure 3.2** An Alternative Representation of Instance $S1$ of Students

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($f_1$:`D1`, ..., $f_n$:`Dn`) 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 `Di`. 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.[2] 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.

| *sid* | *name* | *login* | *age* | *gpa* |
|---|---|---|---|---|
| 50000 | Dave | dave@cs | 19 | 3.2 |
| 53666 | Jones | jones@cs | 18 | 3.3 |
| 53688 | Smith | smith@ee | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.7 |
| 53831 | Madayan | madayan@music | 11 | 1.8 |
| 53832 | Guldu | guldu@music | 12 | 2.0 |

**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

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:[3]

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.

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

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 `UNIQUE` constraint. At most one of these 'candidate' keys can be declared to be a *primary key*, using the `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:

```
CREATE TABLE Students ( sid    CHAR(20),
                        name   CHAR(30),
                        login  CHAR(20),
                        age    INTEGER,
                        gpa    REAL,
                        UNIQUE (name, age),
                        CONSTRAINT StudentsKey PRIMARY KEY (sid) )
```

This definition says that *sid* is the primary key and that the combination of *name* and *age* is also a key. The definition of the primary key also illustrates how we can name a constraint by preceding it with `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 *foreign key* constraint.

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

Enrolled(*sid:* `string`, *cid:* `string`, *grade:* `string`)

To ensure that only bona fide students can enroll in courses, any value that appears in the *sid* field of an instance of the Enrolled relation should also appear in the *sid* field of some tuple in the Students relation. The *sid* field of Enrolled is called a **foreign key** and **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 *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.

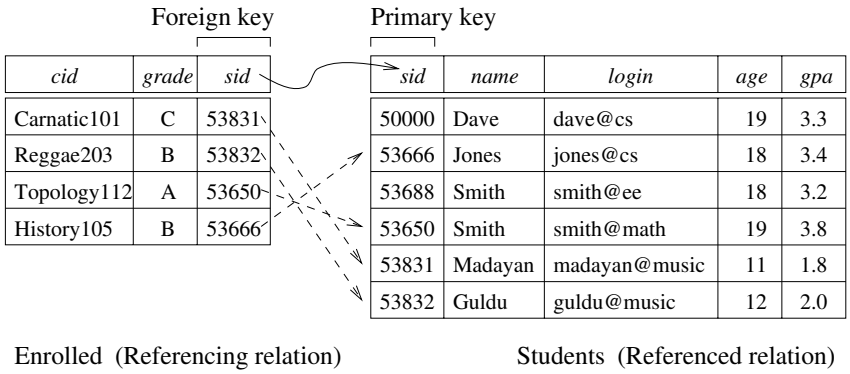



**Figure 3.4** Referential Integrity

If we try to insert the tuple ⟨*55555, Art104, A*⟩ into $E1$, the IC is violated because there is no tuple in $S1$ with the id 55555; the database system should reject such an insertion. Similarly, if we delete the tuple ⟨*53666, Jones, jones@cs, 18, 3.4*⟩ from $S1$, we violate the foreign key constraint because the tuple ⟨*53666, History105, B*⟩ in $E1$ 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 *partner* and declare this column to be a foreign key referring to Students. Intuitively, every student could then have a partner, and the *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 **null**. The use of *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 *null* in a foreign key field does not violate the foreign key constraint. However, *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 *null* values further in Chapter 5.

## Specifying Foreign Key Constraints in SQL-92

Let us define Enrolled(*sid:* `string`, *cid:* `string`, *grade:* `string`):

```
CREATE TABLE Enrolled ( sid    CHAR(20),
```

```
cid    CHAR(20),
grade  CHAR(10),
PRIMARY KEY (sid, cid),
FOREIGN KEY (sid) REFERENCES Students )
```

The foreign key constraint states that every *sid* value in Enrolled must also appear in Students, that is, *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.

| *sid* | *name* | *login* | *age* | *gpa* |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@ee | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |

**Figure 3.5** An Instance $S2$ of the Students Relation

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 *age* values more stringently than is possible by simply using a standard domain such as `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 gpa greater than 3.

Current relational database systems support such general constraints in the form of *table constraints* and *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 $S1$ 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:

```
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*:

```
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:

```
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:

```
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
```