3

THE RELATIONAL MODEL

- How is data represented in the relational model?
- What integrity constraints can be expressed?
- How can data be created and modified?
- How can data be manipulated and queried?
- How can we create, modify, and query tables using SQL?
- How do we obtain a relational database design from an ER diagram?
- What are views and why are they used?
- **Key concepts:** relation, schema, instance, tuple, field, domain, degree, cardinality; SQL DDL, `CREATE TABLE`, `INSERT`, `DELETE`, `UPDATE`; integrity constraints, domain constraints, key constraints, `PRIMARY KEY`, `UNIQUE`, foreign key constraints, `FOREIGN KEY`; referential integrity maintenance, deferred and immediate constraints; relational queries; logical database design, translating ER diagrams to relations, expressing ER constraints using SQL; views, views and logical independence, security; creating views in SQL, updating views, querying views, dropping views

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

SQL. Originally developed as the query language of the pioneering System-R relational DBMS at IBM, structured query language (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 specific to one product from those that are standard; an application that relies on nonstandard 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 (the core subset of) SQL-92 and are working to support the recently adopted SQL:1999 version of the standard, a major extension of SQL-92. Our coverage of SQL is based on SQL:1999, but is applicable to SQL-92 as well; features unique to SQL:1999 are explicitly noted.

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-1970s, and several vendors were offering relational database products shortly thereafter. Today, the relational model is by far the dominant data model and 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.

While we concentrate on the underlying concepts, we also introduce the **Data Definition Language (DDL)** features of SQL, 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 converting an ER diagram into a relational database schema in Section 3.5. 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. In Section 3.7, we describe SQL commands to destroy and alter tables and views.

Finally, in Section 3.8 we extend our design case study, the Internet shop introduced in Section 2.8, by showing how the ER diagram for its conceptual schema can be mapped to the relational model, and how the use of views can help in this design.

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**.

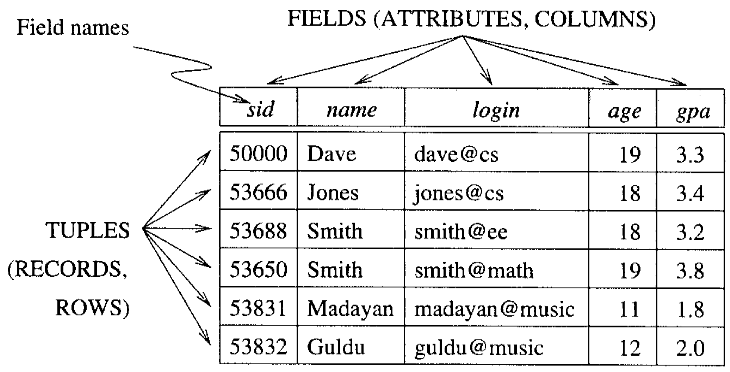
We use the example of student information in a university database from Chapter 1 to illustrate the parts of a relation schema:

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 $S1$



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

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.

In practice, commercial systems allow tables to have duplicate rows, but we assume that a relation is indeed a set of tuples unless otherwise noted. 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 our schema definitions and

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

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 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. Therefore, 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 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

The SQL language standard uses the word *table* to denote *relation*, and we 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 5.7. For now, we only consider domains that are built-in types, such as `integer`.

The `CREATE TABLE` statement is used to define a new table.[1] 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.

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'
```

[1]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 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 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 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 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.) It is important to specify exactly when integrity constraints are checked relative to the statement that causes the change in the data and the transaction that it is part of. We discuss this aspect in Chapter 16, after presenting the transaction concept, which we introduced in Chapter 1, in more detail.

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:[2]

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.

[2]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 on 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

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 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

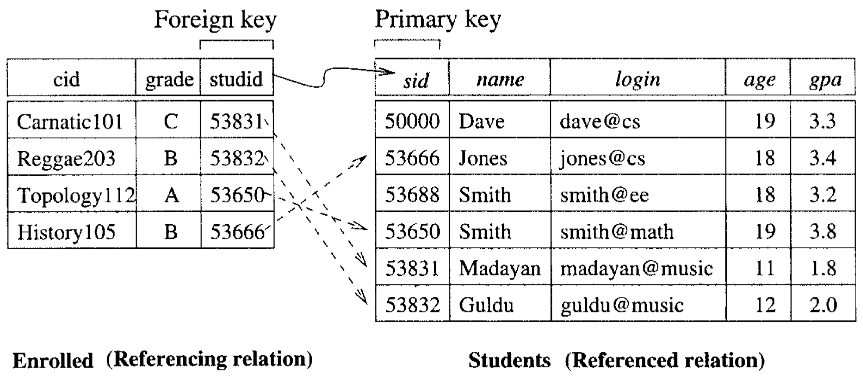
Sometimes the information stored in a relation is linked to the information stored in another relation. If one of the relations is modified, the other must be checked, and perhaps modified, to keep the data consistent. An IC involving both relations must be specified if a DBMS is to make such checks.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

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

Enrolled(*studid:* string, *cid:* string, *grade:* string)

To ensure that only bona fide students can enroll in courses, any value that appears in the *studid* field of an instance of the Enrolled relation should also appear in the *sid* field of some tuple in the Students relation. The *studid* 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); that is, 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 tuples that are not referenced from Enrolled (e.g., the student with *sid=50000*). However, every *studid* value that appears in the instance of the Enrolled table appears in the primary key column of a row in the Students table.


Foreign key
Primary key
Enrolled (Referencing relation)
Students (Referenced relation)

cid	grade	studid
Carnatic101	C	53831
Reggae203	B	53832
Topology112	A	53650
History105	B	53666

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 3.4 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 *sid* 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 *studid* 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 discuss *null* values further in Chapter 5.

Specifying Foreign Key Constraints in SQL

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

```
CREATE TABLE Enrolled ( studid CHAR(20),
                        cid    CHAR(20),
                        grade  CHAR(10),
                        PRIMARY KEY (studid, cid),
                        FOREIGN KEY (studid) REFERENCES Students )
```

The foreign key constraint states that every *studid* value in Enrolled must also appear in Students, that is, *studid* in Enrolled is a foreign key referencing Students. Specifically, every *studid* value in Enrolled must appear as the value in the primary key field, *sid*, of Students. Incidentally, the primary key constraint for Enrolled states that a student has exactly one grade for each course 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 rejects 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 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 5.7 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. Every 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 Section 3.3.1.

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, that is, 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 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 *S*1 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 Students tuple with *sid* 51111:

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

On the other hand, insertions of Students tuples do not violate referential integrity, and deletions of Students tuples could cause violations. Further, updates on either Enrolled or Students that change the *studid* (respectively, *sid*) value could potentially violate referential integrity.

SQL 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* studid *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 *studid* 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 *studid* column to *null*. In our example, this option conflicts with the fact that *studid* is part of the primary key of Enrolled and therefore cannot be set to *null*. Therefore, we are limited to the first three options in our example, although this fourth option (setting the foreign key to *null*) is available in general.

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 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 ( studid CHAR(20),
                        cid    CHAR(20),
                        grade  CHAR(10),
                        PRIMARY KEY (studid, cid),
                        FOREIGN KEY (studid) 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.

3.3.1 Transactions and Constraints

As we saw in Chapter 1, a program that runs against a database is called a transaction, and it can contain several statements (queries, inserts, updates, etc.) that access the database. If (the execution of) a statement in a transaction violates an integrity constraint, should the DBMS detect this right away or should all constraints be checked together just before the transaction completes?

By default, a constraint is checked at the end of every SQL statement that could lead to a violation, and if there is a violation, the statement is rejected. Sometimes this approach is too inflexible. Consider the following variants of the Students and Courses relations; every student is required to have an honors course, and every course is required to have a grader, who is some student.

```
CREATE TABLE Students ( sid    CHAR(20),
                        name   CHAR(30),
                        login  CHAR(20),
                        age    INTEGER,
                        honorsCHAR(10) NOT NULL,
                        gpa    REAL )
                        PRIMARY KEY (sid),
                        FOREIGN KEY (honors) REFERENCES Courses (cid))

CREATE TABLE Courses (  cid    CHAR(10),
                        cname CHAR(10),
                        creditsINTEGER,
                        grader CHAR(20) NOT NULL,
                        PRIMARY KEY (cid)
                        FOREIGN KEY (grader) REFERENCES Students (sid))
```

Whenever a Students tuple is inserted, a check is made to see if the honors course is in the Courses relation, and whenever a Courses tuple is inserted, a check is made to see that the grader is in the Students relation. How are we to insert the very first course or student tuple? One cannot be inserted without the other. The only way to accomplish this insertion is to **defer** the constraint checking that would normally be carried out at the end of an INSERT statement.

SQL allows a constraint to be in DEFERRED or IMMEDIATE mode.

```
SET CONSTRAINT ConstraintFoo DEFERRED
```

A constraint in deferred mode is checked at commit time. In our example, the foreign key constraints on Boats and Sailors can both be declared to be in deferred mode. We can then insert a boat with a nonexistent sailor as the captain (temporarily making the database inconsistent), insert the sailor (restoring consistency), then commit and check that both constraints are satisfied.

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. 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.

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.6 Students with *age* < 18 on Instance *S*1

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 produces 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:

```
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.

name	*login*
Madayan	madayan@music
Guldu	guldu@music

Figure 3.7 Names and Logins of Students under 18

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:

```
SELECT S.name, E.cid
FROM   Students S, Enrolled E
WHERE  S.sid = E.studid AND E.grade = 'A'
```

This query can be understood as follows: "If there is a Students tuple S and an Enrolled tuple E such that S.sid = E.studid (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 cover relational queries and SQL 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, a standard approach is taken 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, unless we use certain SQL constraints that are costly to check.) We now describe how to translate an ER diagram into a collection of tables with associated constraints, that is, 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

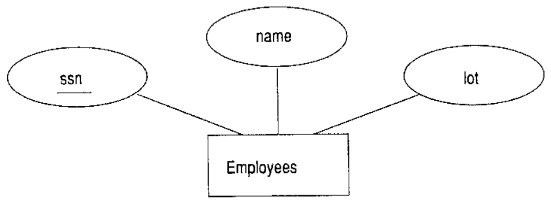


Figure 3.8 The Employees Entity Set

Employees entities, is shown in Figure 3.9 in a tabular format.

ssn	*name*	*lot*
123-22-3666	Attishoo	48
231-31-5368	Smiley	22
131-24-3650	Smethurst	35

Figure 3.9 An Instance of the Employees Entity Set

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

```
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.

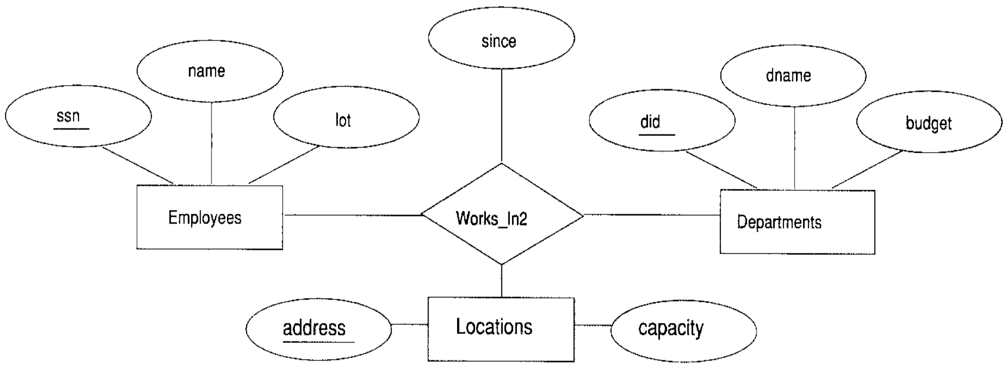


Figure 3.10 A Ternary Relationship Set

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

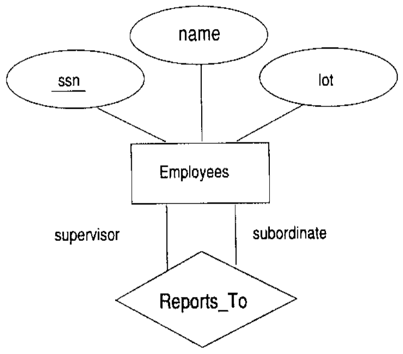


Figure 3.11 The Reports_To Relationship Set

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. Hence 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 cor-

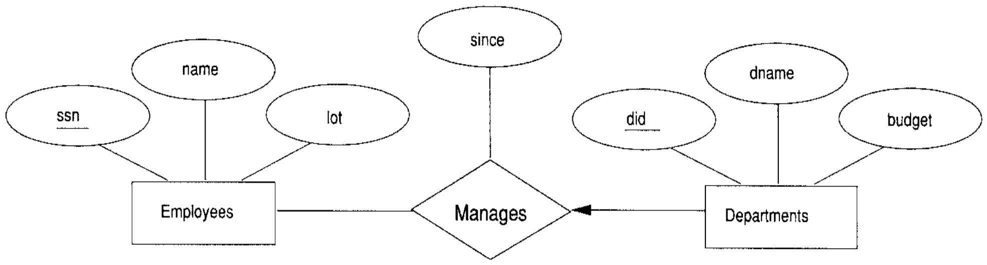


Figure 3.12 Key Constraint on Manages

responding to Manages has the attributes *ssn, did, since*. However, because each department has at most one manager, no two tuples can have the same *did* value but differ on the *ssn* value. A consequence of this observation is that *did* is itself a key for Manages; indeed, the set *did, ssn* is not a key (because it is not minimal). The Manages relation can be defined using the following SQL statement:

```
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 *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 *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:

```
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.

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:

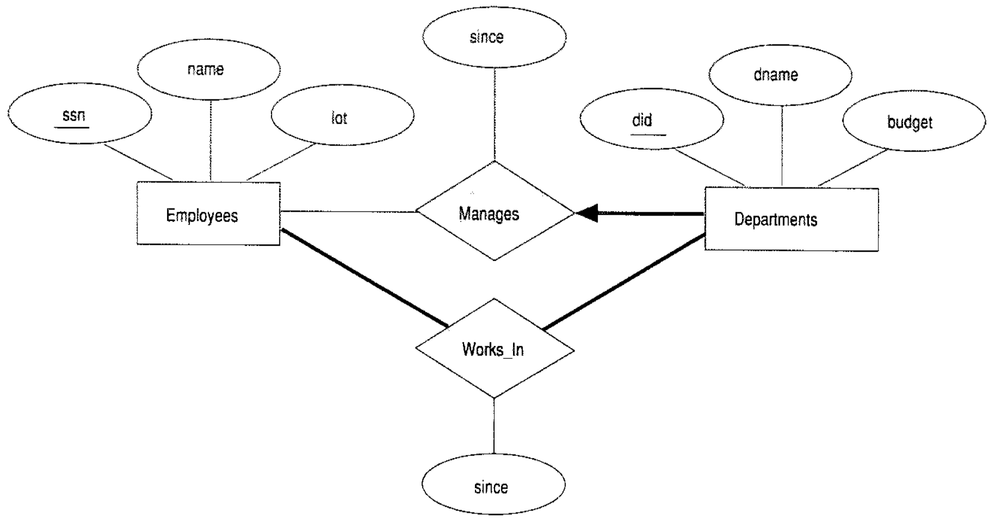


Figure 3.13 Manages and Works_In

```
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, 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 5.7) 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, 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 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.

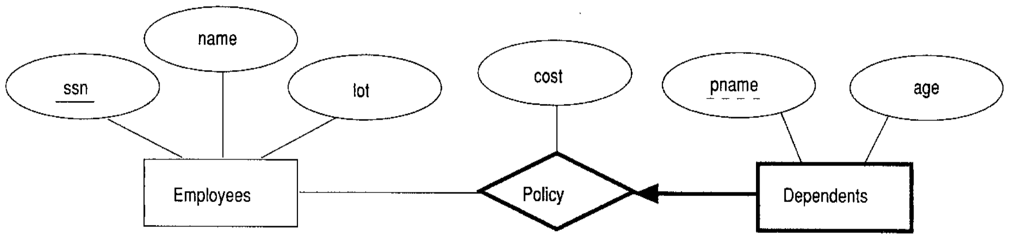


Figure 3.14 The Dependents Weak Entity Set

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

```
CREATE TABLE Dep_Policy ( pname  CHAR(20),
                          age    INTEGER,
                          cost   REAL,
                          ssn    CHAR(11),
                          PRIMARY KEY (pname, ssn),
                          FOREIGN KEY (ssn) REFERENCES Employees
                                 ON DELETE CASCADE )
```

Observe that the primary key is ⟨*pname*, *ssn*⟩, 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, *ssn* cannot be *null*. This is ensured because *ssn* is part of the primary key. The 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:

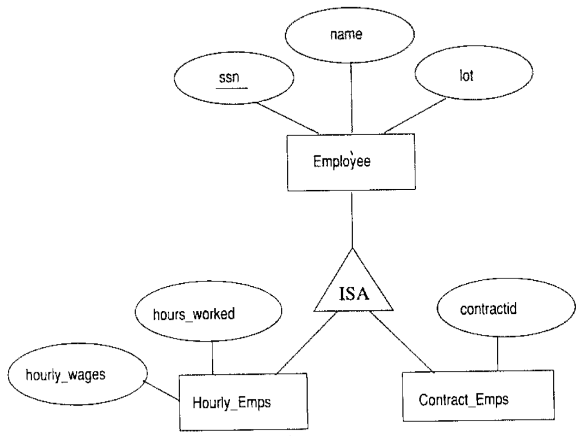


Figure 3.15 Class Hierarchy

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 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 19. 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 only by using assertions.

3.5.7 Translating ER Diagrams with Aggregation

Consider the ER diagram shown in Figure 3.16. The Employees, Projects,

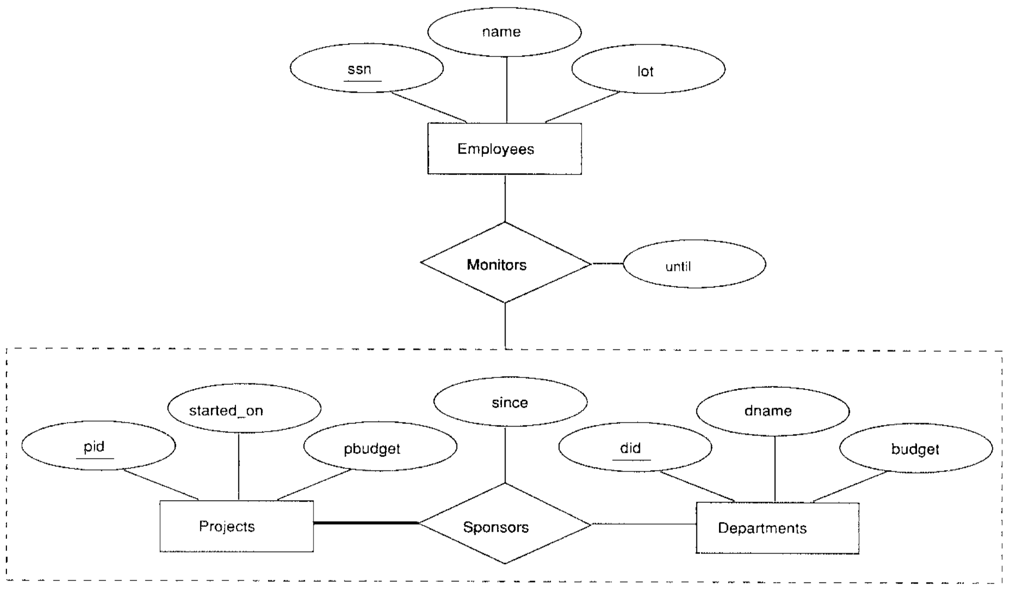


Figure 3.16 Aggregation

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 Sponsors (*did, pid*), and the descriptive attributes of Monitors (*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 by dropping the Sponsors relation. Consider the Sponsors relation. It has attributes *pid, did,* 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, *since*) of the Sponsors relationship.
2. Not every sponsorship has a monitor, and thus some ⟨*pid, did*⟩ 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 from the ⟨*pid, did*⟩ columns of Monitors; Sponsors can be dropped.

3.5.8 ER to Relational: Additional Examples

Consider the ER diagram shown in Figure 3.17. We can use the key constraints

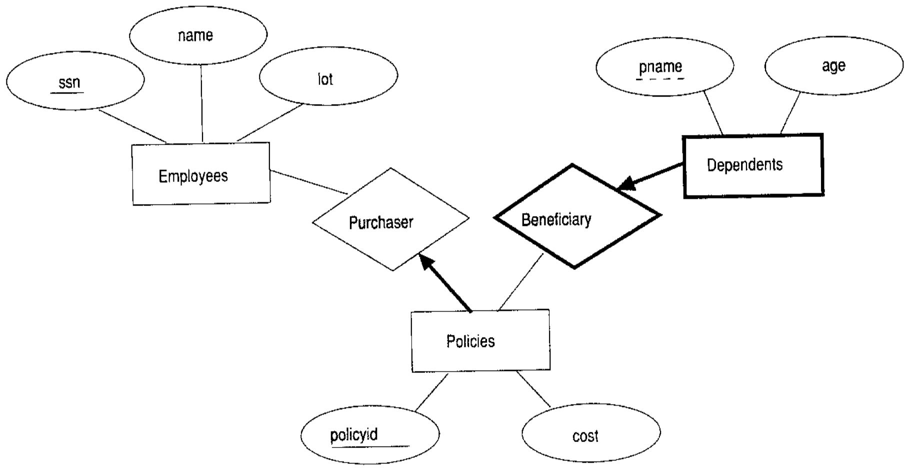


Figure 3.17 Policy Revisited

to combine Purchaser information with Policies and Beneficiary information with Dependents, and translate it into the relational model as follows:

```
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* distinguishes only 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 ⟨*policyid, ssn*⟩, 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:

```
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 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 course identifier. We can define a view for this purpose. Using SQL notation:

```
    CREATE VIEW B-Students (name, sid, course)
           AS SELECT S.sname, S.sid, E.cid
```

```
FROM    Students S, Enrolled E
WHERE   S.sid = E.studid 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, then the rest of the query is evaluated treating B-Students like any other relation referred to in the query. (We discuss how queries on views are evaluated in practice in Chapter 25.)

name	*sid*	*course*
Jones	53666	History105
Guldu	53832	Reggae203

Figure 3.18 An Instance of the B-Students View

3.6.1 Views, Data Independence, Security

Consider the levels of abstraction 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, that is, 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 the other students' name and age but not their gpa, and allows all students to access this view but not the underlying Students table (see Chapter 21).

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.[3] 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 then affects 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.

[3]There is also the restriction that the DISTINCT operator cannot be used in updatable view definitions. By default, SQL does not eliminate duplicate copies of rows from the result of a query; the DISTINCT operator requires duplicate elimination. We discuss this point further in Chapter 5.

Updatable Views in SQL:1999 The new SQL standard has expanded the class of view definitions that are updatable, taking primary key constraints into account. In contrast to SQL-92, a view definition that contains more than one table in the `FROM` clause may be updatable under the new definition. Intuitively, we can update a field of a view if it is obtained from exactly one of the underlying tables, and the primary key of that table is included in the fields of the view.

SQL:1999 distinguishes between views whose rows can be modified (*updatable views*) and views into which new rows can be inserted (**insertable-into views**): Views defined using the SQL constructs `UNION`, `INTERSECT`, and `EXCEPT` (which we discuss in Chapter 5) cannot be inserted into, even if they are updatable. Intuitively, updatability ensures that an updated tuple in the view can be traced to exactly one tuple in one of the tables used to define the view. The updatability property, however, may still not enable us to decide into which table to insert a new tuple.

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 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. In this case, only rows that will actually appear in the view are permissible insertions.

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 not go into the details.

Need to Restrict View Updates

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

Clubs(*cname:* `string`, *jyear:* `date`, *mname:* `string`)

cname	*jyear*	*mname*
Sailing	1996	Dave
Hiking	1997	Smith
Rowing	1998	Smith

Figure 3.19 An Instance *C* of Clubs

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

Figure 3.20 An Instance *S*3 of Students

name	*login*	*club*	*since*
Dave	dave@cs	Sailing	1996
Smith	smith@ee	Hiking	1997
Smith	smith@ee	Rowing	1998
Smith	smith@math	Hiking	1997
Smith	smith@math	Rowing	1998

Figure 3.21 Instance of ActiveStudents

A tuple in Clubs denotes that the student called *mname* has been a member of the club *cname* since the date *jyear*.[4] Suppose that we are often interested in finding the names and logins of students with a 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:

```
CREATE VIEW ActiveStudents (name, login, club, since)
       AS SELECT S.sname, S.login, C.cname, C.jyear
          FROM   Students S, Clubs C
          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 *S*3, ActiveStudents contains the rows shown in Figure 3.21.

Now suppose that we want to delete the row ⟨*Smith, smith@ee, Hiking, 1997*⟩ from ActiveStudents. How are we to do this? ActiveStudents rows are not stored explicitly but 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 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*⟩

[4]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.

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 side effect is desirable. In fact, the only reasonable solution is to *disallow* such updates on views.

Views involving more than one base table can, in principle, be safely updated. The B-Students view 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 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 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 `RESTRICT` destroys the Students table unless some view or integrity constraint refers to Students; if so, the command fails. If the keyword `RESTRICT` is replaced by `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:

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

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 add or drop integrity constraints on a table; we do not discuss these aspects of the command beyond remarking that dropping columns is treated very similarly to dropping tables or views.

3.8 CASE STUDY: THE INTERNET STORE

The next design step in our running example, continued from Section 2.8, is logical database design. Using the standard approach discussed in Chapter 3, DBDudes maps the ER diagram shown in Figure 2.20 to the relational model, generating the following tables:

```
CREATE TABLE Books ( isbn           CHAR(10),
                     title          CHAR(80),
                     author         CHAR(80),
                     qty_in_stock   INTEGER,
                     price          REAL,
                     year_published INTEGER,
                     PRIMARY KEY (isbn))

CREATE TABLE Orders ( isbn        CHAR(10),
                      cid         INTEGER,
                      cardnum     CHAR(16),
                      qty         INTEGER,
                      order_date  DATE,
                      ship_date   DATE,
                      PRIMARY KEY (isbn,cid),
                      FOREIGN KEY (isbn) REFERENCES Books,
                      FOREIGN KEY (cid) REFERENCES Customers )

CREATE TABLE Customers ( cid      INTEGER,
                         cname    CHAR(80),
                         address  CHAR(200),
                         PRIMARY KEY (cid)
```

The design team leader, who is still brooding over the fact that the review exposed a flaw in the design, now has an inspiration. The Orders table contains the field *order_date* and the key for the table contains only the fields *isbn* and *cid*. Because of this, a customer cannot order the same book on different days, a restriction that was not intended. Why not add the *order_date* attribute to the key for the Orders table? This would eliminate the unwanted restriction:

```
CREATE TABLE Orders (    isbn        CHAR(10),
                         ...
                         PRIMARY KEY (isbn,cid,ship_date),
                         ...)
```

The reviewer, Dude 2, is not entirely happy with this solution, which he calls a 'hack'. He points out that no natural ER diagram reflects this design and stresses the importance of the ER diagram as a design document. Dude 1 argues that, while Dude 2 has a point, it is important to present B&N with a preliminary design and get feedback; everyone agrees with this, and they go back to B&N.

The owner of B&N now brings up some additional requirements he did not mention during the initial discussions: "Customers should be able to purchase several different books in a single order. For example, if a customer wants to purchase three copies of 'The English Teacher' and two copies of 'The Character of Physical Law,' the customer should be able to place a single order for both books."

The design team leader, Dude 1, asks how this affects the shippping policy. Does B&N still want to ship all books in an order together? The owner of B&N explains their shipping policy: "As soon as we have have enough copies of an ordered book we ship it, even if an order contains several books. So it could happen that the three copies of 'The English Teacher' are shipped today because we have five copies in stock, but that 'The Character of Physical Law' is shipped tomorrow, because we currently have only one copy in stock and another copy arrives tomorrow. In addition, my customers could place more than one order per day, and they want to be able to identify the orders they placed."

The DBDudes team thinks this over and identifies two new requirements: First, it must be possible to order several different books in a single order and second, a customer must be able to distinguish between several orders placed the same day. To accomodate these requirements, they introduce a new attribute into the Orders table called *ordernum*, which uniquely identifies an order and therefore the customer placing the order. However, since several books could be purchased in a single order, *ordernum* and *isbn* are both needed to determine *qty* and *ship_date* in the Orders table.

Orders are assigned order numbers sequentially and orders that are placed later have higher order numbers. If several orders are placed by the same customer on a single day, these orders have different order numbers and can thus be distinguished. The SQL DDL statement to create the modified Orders table follows:

```
CREATE TABLE Orders ( ordernum     INTEGER,
                      isbn         CHAR(10),
                      cid          INTEGER,
                      cardnum      CHAR(16),
                      qty          INTEGER,
                      order_date   DATE,
                      ship_date    DATE,
                      PRIMARY KEY (ordernum, isbn),
                      FOREIGN KEY (isbn) REFERENCES Books
                      FOREIGN KEY (cid) REFERENCES Customers )
```

The owner of B&N is quite happy with this design for Orders, but has realized something else. (DBDudes is not surprised; customers almost always come up with several new requirements as the design progresses.) While he wants all his employees to be able to look at the details of an order, so that they can respond to customer enquiries, he wants customers' credit card information to be secure. To address this concern, DBDudes creates the following view:

```
CREATE VIEW OrderInfo (isbn, cid, qty, order_date, ship_date)
       AS SELECT O.cid, O.qty, O.order_date, O.ship_date
          FROM   Orders O
```

The plan is to allow employees to see this table, but not Orders; the latter is restricted to B&N's Accounting division. We'll see how this is accomplished in Section 21.7.

3.9 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What is a relation? Differentiate between a relation schema and a relation instance. Define the terms *arity* and *degree* of a relation. What are domain constraints? **(Section 3.1)**
- What SQL construct enables the definition of a relation? What constructs allow modification of relation instances? **(Section 3.1.1)**
- What are *integrity constraints*? Define the terms *primary key constraint* and *foreign key constraint*. How are these constraints expressed in SQL? What other kinds of constraints can we express in SQL? **(Section 3.2)**

- What does the DBMS do when constraints are violated? What is *referential integrity*? What options does SQL give application programmers for dealing with violations of referential integrity? **(Section 3.3)**
- When are integrity constraints enforced by a DBMS? How can an application programmer control the time that constraint violations are checked during transaction execution? **(Section 3.3.1)**
- What is a *relational database query*? **(Section 3.4)**
- How can we translate an ER diagram into SQL statements to create tables? How are entity sets mapped into relations? How are relationship sets mapped? How are constraints in the ER model, weak entity sets, class hierarchies, and aggregation handled? **(Section 3.5)**
- What is a *view*? How do views support logical data independence? How are views used for security? How are queries on views evaluated? Why does SQL restrict the class of views that can be updated? **(Section 3.6)**
- What are the SQL constructs to modify the structure of tables and destroy tables and views? Discuss what happens when we destroy a view. **(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 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:

> Emp(*eid:* `integer`, *ename:* `string`, *age:* `integer`, *salary:* `real`)
> Works(*eid:* `integer`, *did:* `integer`, *pct_time:* `integer`)
> Dept(*did:* `integer`, *dname:* `string`, *budget:* `real`, *managerid:* `integer`)

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 preceding 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 percent 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 $a1$ and $a2$, with $a1$ 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 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.

Exercise 3.19 Briefly answer the following questions based on this schema:

Emp(<u>*eid:* `integer`</u>, *ename:* `string`, *age:* `integer`, *salary:* `real`)
Works(<u>*eid:* `integer`, *did:* `integer`</u>, *pct_time:* `integer`)
Dept(<u>*did:* `integer`</u>, *budget:* `real`, *managerid:* `integer`)

1. Suppose you have a view SeniorEmp defined as follows:

```
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:

```
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.

Exercise 3.20 Consider the following schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *address:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Catalog(*sid:* `integer`, *pid:* `integer`, *cost:* `real`)

The Catalog relation lists the prices charged for parts by Suppliers. Answer the following questions:

- Give an example of an updatable view involving one relation.
- Give an example of an updatable view involving two relations.
- Give an example of an insertable-into view that is updatable.
- Give an example of an insertable-into view that is not updatable.

PROJECT-BASED EXERCISES

Exercise 3.21 Create the relations Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets_In in Minibase.

Exercise 3.22 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 3.23 What integrity constraints are enforced by Minibase?

Exercise 3.24 Run the SQL queries presented in this chapter.

BIBLIOGRAPHIC NOTES

The relational model was proposed in a seminal paper by Codd [187]. Childs [176] and Kuhns [454] foreshadowed some of these developments. Gallaire and Minker's book [296] 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 [746]. Extensions of the relational model to incorporate *null* values, which indicate an unknown or missing field value, are discussed by several authors; for example, [329, 396, 622, 754, 790].

Pioneering projects include System R [40, 150] at IBM San Jose Research Laboratory (now IBM Almaden Research Center), Ingres [717] at the University of California at Berkeley, PRTV [737] at the IBM UK Scientific Center in Peterlee, and QBE [801] 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 [45]; Maier [501]; and Abiteboul, Hull, and Vianu [3]. [415] is an excellent survey article.

Integrity constraints in relational databases have been discussed at length. [190] addresses semantic extensions to the relational model, and integrity, in particular referential integrity. [360] discusses semantic integrity constraints. [203] 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. [51] compares the cost of enforcing integrity constraints via compile-time, run-time, and post-execution checks. [145] presents an SQL-based language for specifying integrity constraints and identifies conditions under which integrity rules specified in this language can be violated. [713] discusses the technique of integrity constraint checking by query modification. [180] discusses real-time integrity constraints. Other papers on checking integrity constraints in databases include [82, 122, 138, 517]. [681] 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 [296] and [295].

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 discuss each of the major commercial systems; for example, Chamberlin's book on DB2 [149], Date and McGoveran's book on Sybase [206], and Koch and Loney's book on Oracle [443].

Several papers consider the problem of translating updates specified on views into updates on the underlying table [59, 208, 422, 468, 778]. [292] is a good survey on this topic. See the bibliographic notes for Chapter 25 for references to work querying views and maintaining materialized views.

[731] 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 [513, 514].

4

RELATIONAL ALGEBRA AND CALCULUS

- What is the foundation for relational query languages like SQL? What is the difference between procedural and declarative languages?
- What is relational algebra, and why is it important?
- What are the basic algebra operators, and how are they combined to write complex queries?
- What is relational calculus, and why is it important?
- What subset of mathematical logic is used in relational calculus, and how is it used to write queries?
- **Key concepts:** relational algebra, select, project, union, intersection, cross-product, join, division; tuple relational calculus, domain relational calculus, formulas, universal and existential quantifiers, bound and free variables

Stand firm in your refusal to remain conscious during algebra. In real life, I assure you, there is no such thing as algebra.

—Fran Lebowitz, *Social Studies*

This chapter presents two formal query languages associated with the relational model. **Query languages** are specialized languages for asking questions, or **queries**, that involve the data in a database. After covering some preliminaries in Section 4.1, we discuss *relational algebra* in Section 4.2. Queries in relational algebra are composed using a collection of operators, and each query describes a step-by-step procedure for computing the desired answer; that is, queries are specified in an *operational* manner. In Section 4.3, we discuss *relational calculus*, in which a query describes the desired answer without specifying how the answer is to be computed; this nonprocedural style of querying is called *declarative*. We usually refer to relational algebra and relational calculus as algebra and calculus, respectively. We compare the expressive power of algebra and calculus in Section 4.4. These formal query languages have greatly influenced commercial query languages such as SQL, which we discuss in later chapters.

4.1 PRELIMINARIES

We begin by clarifying some important points about relational queries. The inputs and outputs of a query are relations. A query is evaluated using *instances* of each input relation and it produces an instance of the output relation. In Section 3.4, we used field names to refer to fields because this notation makes queries more readable. An alternative is to always list the fields of a given relation in the same order and refer to fields by position rather than by field name.

In defining relational algebra and calculus, the alternative of referring to fields by position is more convenient than referring to fields by name: Queries often involve the computation of intermediate results, which are themselves relation instances; and if we use field names to refer to fields, the definition of query language constructs must specify the names of fields for all intermediate relation instances. This can be tedious and is really a secondary issue, because we can refer to fields by position anyway. On the other hand, field names make queries more readable.

Due to these considerations, we use the positional notation to formally define relational algebra and calculus. We also introduce simple conventions that allow intermediate relations to 'inherit' field names, for convenience.

We present a number of sample queries using the following schema:

Sailors(*sid:* `integer`, *sname:* `string`, *rating:* `integer`, *age:* `real`)
Boats(*bid:* `integer`, *bname:* `string`, *color:* `string`)
Reserves(*sid:* `integer`, *bid:* `integer`, *day:* `date`)

The key fields are underlined, and the domain of each field is listed after the field name. Thus, *sid* is the key for Sailors, *bid* is the key for Boats, and all three fields together form the key for Reserves. Fields in an instance of one of these relations are referred to by name, or positionally, using the order in which they were just listed.

In several examples illustrating the relational algebra operators, we use the instances $S1$ and $S2$ (of Sailors) and $R1$ (of Reserves) shown in Figures 4.1, 4.2, and 4.3, respectively.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Figure 4.1 Instance $S1$ of Sailors

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

Figure 4.2 Instance $S2$ of Sailors

sid	*bid*	*day*
22	101	10/10/96
58	103	11/12/96

Figure 4.3 Instance $R1$ of Reserves

4.2 RELATIONAL ALGEBRA

Relational algebra is one of the two formal query languages associated with the relational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) relation instances as arguments and returns a relation instance as the result. This property makes it easy to *compose* operators to form a complex query—a **relational algebra expression** is recursively defined to be a relation, a unary algebra operator applied to a single expression, or a binary algebra operator applied to two expressions. We describe the basic operators of the algebra (selection, projection, union, cross-product, and difference), as well as some additional operators that can be defined in terms of the basic operators but arise frequently enough to warrant special attention, in the following sections.

Each relational query describes a step-by-step procedure for computing the desired answer, based on the order in which operators are applied in the query. The procedural nature of the algebra allows us to think of an algebra expression as a recipe, or a plan, for evaluating a query, and relational systems in fact use algebra expressions to represent query evaluation plans.

4.2.1 Selection and Projection

Relational algebra includes operators to *select* rows from a relation (σ) and to *project* columns (π). These operations allow us to manipulate data in a single relation. Consider the instance of the Sailors relation shown in Figure 4.2, denoted as *S2*. We can retrieve rows corresponding to expert sailors by using the σ operator. The expression

$$\sigma_{rating>8}(S2)$$

evaluates to the relation shown in Figure 4.4. The subscript *rating>8* specifies the selection criterion to be applied while retrieving tuples.

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
58	Rusty	10	35.0

Figure 4.4 $\sigma_{rating>8}(S2)$

sname	*rating*
yuppy	9
Lubber	8
guppy	5
Rusty	10

Figure 4.5 $\pi_{sname,rating}(S2)$

The selection operator σ specifies the tuples to retain through a *selection condition.* In general, the selection condition is a Boolean combination (i.e., an expression using the logical connectives $\wedge$ and $\vee$) of *terms* that have the form *attribute* op *constant* or *attribute1* op *attribute2*, where op is one of the comparison operators $<, <=, =, \neq, >=$, or $>$. The reference to an attribute can be by position (of the form *.i* or *i*) or by name (of the form *.name* or *name*). The schema of the result of a selection is the schema of the input relation instance.

The projection operator π allows us to extract columns from a relation; for example, we can find out all sailor names and ratings by using π. The expression

$$\pi_{sname,rating}(S2)$$

evaluates to the relation shown in Figure 4.5. The subscript *sname,rating* specifies the fields to be retained; the other fields are 'projected out.' The schema of the result of a projection is determined by the fields that are projected in the obvious way.

Suppose that we wanted to find out only the ages of sailors. The expression

$$\pi_{age}(S2)$$

evaluates to the relation shown in Figure 4.6. The important point to note is that, although three sailors are aged 35, a single tuple with *age=35.0* appears in the result of the projection. This follows from the definition of a relation as a *set* of tuples. In practice, real systems often omit the expensive step of eliminating *duplicate tuples*, leading to relations that are multisets. However, our discussion of relational algebra and calculus assumes that duplicate elimination is always done so that relations are always sets of tuples.

Since the result of a relational algebra expression is always a relation, we can substitute an expression wherever a relation is expected. For example, we can compute the names and ratings of highly rated sailors by combining two of the preceding queries. The expression

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

produces the result shown in Figure 4.7. It is obtained by applying the selection to $S2$ (to get the relation shown in Figure 4.4) and then applying the projection.

age
35.0
55.5

Figure 4.6 $\pi_{age}(S2)$

sname	*rating*
yuppy	9
Rusty	10

Figure 4.7 $\pi_{sname,rating}(\sigma_{rating>8}(S2))$

4.2.2 Set Operations

The following standard operations on sets are also available in relational algebra: *union* ($\cup$), *intersection* ($\cap$), *set-difference* ($-$), and *cross-product* ($\times$).

- **Union:** $R \cup S$ returns a relation instance containing all tuples that occur in *either* relation instance R or relation instance S (or both). R and S must be *union-compatible*, and the schema of the result is defined to be identical to the schema of R.

  Two relation instances are said to be **union-compatible** if the following conditions hold:
  - they have the same number of the fields, and
  - corresponding fields, taken in order from left to right, have the same *domains*.

  Note that field names are not used in defining union-compatibility. For convenience, we will assume that the fields of $R \cup S$ inherit names from R, if the fields of R have names. (This assumption is implicit in defining the schema of $R \cup S$ to be identical to the schema of R, as stated earlier.)

- **Intersection:** $R \cap S$ returns a relation instance containing all tuples that occur in *both* R and S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

- **Set-difference:** $R - S$ returns a relation instance containing all tuples that occur in R but not in S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

- **Cross-product:** $R \times S$ returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S). The result of $R \times S$ contains one tuple $\langle r,\ s\rangle$ (the concatenation of tuples r and s) for each pair of tuples $r \in R,\ s \in S$. The cross-product opertion is sometimes called **Cartesian product**.

  We use the convention that the fields of $R \times S$ inherit names from the corresponding fields of R and S. It is possible for both R and S to contain one or more fields having the same name; this situation creates a *naming conflict*. The corresponding fields in $R \times S$ are unnamed and are referred to solely by position.

In the preceding definitions, note that each operator can be applied to relation instances that are computed using a relational algebra (sub)expression.

We now illustrate these definitions through several examples. The union of $S1$ and $S2$ is shown in Figure 4.8. Fields are listed in order; field names are also inherited from $S1$. $S2$ has the same field names, of course, since it is also an instance of Sailors. In general, fields of $S2$ may have different names; recall that we require only domains to match. Note that the result is a *set* of tuples. Tuples that appear in both $S1$ and $S2$ appear only once in $S1 \cup S2$. Also, $S1 \cup R1$ is not a valid operation because the two relations are not union-compatible. The intersection of $S1$ and $S2$ is shown in Figure 4.9, and the set-difference $S1 - S2$ is shown in Figure 4.10.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0
28	yuppy	9	35.0
44	guppy	5	35.0

Figure 4.8 $S1 \cup S2$

The result of the cross-product $S1 \times R1$ is shown in Figure 4.11. Because $R1$ and $S1$ both have a field named *sid*, by our convention on field names, the corresponding two fields in $S1 \times R1$ are unnamed, and referred to solely by the position in which they appear in Figure 4.11. The fields in $S1 \times R1$ have the same domains as the corresponding fields in $R1$ and $S1$. In Figure 4.11, *sid* is

sid	*sname*	*rating*	*age*
31	Lubber	8	55.5
58	Rusty	10	35.0

Figure 4.9 $S1 \cap S2$

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0

Figure 4.10 $S1 - S2$

listed in parentheses to emphasize that it is not an inherited field name; only the corresponding domain is inherited.

(*sid*)	*sname*	*rating*	*age*	(*sid*)	*bid*	*day*
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

Figure 4.11 $S1 \times R1$

4.2.3 Renaming

We have been careful to adopt field name conventions that ensure that the result of a relational algebra expression inherits field names from its argument (input) relation instances in a natural way whenever possible. However, name conflicts can arise in some cases; for example, in $S1 \times R1$. It is therefore convenient to be able to give names explicitly to the fields of a relation instance that is defined by a relational algebra expression. In fact, it is often convenient to give the instance itself a name so that we can break a large algebra expression into smaller pieces by giving names to the results of subexpressions.

We introduce a **renaming** operator ρ for this purpose. The expression $\rho(R(\overline{F}), E)$ takes an arbitrary relational algebra expression E and returns an instance of a (new) relation called R. R contains the same tuples as the result of E and has the same schema as E, but some fields are renamed. The field names in relation R are the same as in E, except for fields renamed in the *renaming list* $\overline{F}$, which is a list of terms having the form *oldname* $\rightarrow$ *newname* or *position* $\rightarrow$ *newname*. For ρ to be well-defined, references to fields (in the form of *oldnames* or *positions* in the renaming list) may be unambiguous and no two fields in the result may have the same name. Sometimes we want to only rename fields or (re)name the relation; we therefore treat both R and $\overline{F}$ as optional in the use of ρ. (Of course, it is meaningless to omit both.)

For example, the expression $\rho(C(1 \rightarrow sid1, 5 \rightarrow sid2),\ S1 \times R1)$ returns a relation that contains the tuples shown in Figure 4.11 and has the following schema: C(*sid1:* `integer`, *sname:* `string`, *rating:* `integer`, *age:* `real`, *sid2:* `integer`, *bid:* `integer`, *day:* `dates`).

It is customary to include some additional operators in the algebra, but all of them can be defined in terms of the operators we have defined thus far. (In fact, the renaming operator is needed only for syntactic convenience, and even the $\cap$ operator is redundant; $R \cap S$ can be defined as $R - (R - S)$.) We consider these additional operators and their definition in terms of the basic operators in the next two subsections.

4.2.4 Joins

The *join* operation is one of the most useful operations in relational algebra and the most commonly used way to combine information from two or more relations. Although a join can be defined as a cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross-products. Further, the result of a cross-product is typically much larger than the result of a join, and it is very important to recognize joins and implement them without materializing the underlying cross-product (by applying the selections and projections 'on-the-fly'). For these reasons, joins have received a lot of attention, and there are several variants of the join operation.[1]

Condition Joins

The most general version of the join operation accepts a *join condition c* and a pair of relation instances as arguments and returns a relation instance. The *join condition* is identical to a *selection condition* in form. The operation is defined as follows:

$$R \bowtie_c S \ = \ \sigma_c(R \times S)$$

Thus $\bowtie$ is defined to be a cross-product followed by a selection. Note that the condition c can (and typically *does*) refer to attributes of both R and S. The reference to an attribute of a relation, say, R, can be by position (of the form $R.i$) or by name (of the form $R.name$).

As an example, the result of $S1 \bowtie_{S1.sid<R1.sid} R1$ is shown in Figure 4.12. Because *sid* appears in both $S1$ and $R1$, the corresponding fields in the result of the cross-product $S1 \times R1$ (and therefore in the result of $S1 \bowtie_{S1.sid<R1.sid} R1$)

[1]Several variants of joins are not discussed in this chapter. An important class of joins, called *outer joins*, is discussed in Chapter 5.

are unnamed. Domains are inherited from the corresponding fields of $S1$ and $R1$.

(sid)	*sname*	*rating*	*age*	*(sid)*	*bid*	*day*
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	58	103	11/12/96

Figure 4.12 $S1 \bowtie_{S1.sid<R1.sid} R1$

Equijoin

A common special case of the join operation $R \bowtie S$ is when the *join condition* consists solely of equalities (connected by $\wedge$) of the form $R.name1 = S.name2$, that is, equalities between two fields in R and S. In this case, obviously, there is some redundancy in retaining both attributes in the result. For join conditions that contain only such equalities, the join operation is refined by doing an additional projection in which $S.name2$ is dropped. The join operation with this refinement is called **equijoin**.

The schema of the result of an equijoin contains the fields of R (with the same names and domains as in R) followed by the fields of S that do not appear in the join conditions. If this set of fields in the result relation includes two fields that inherit the same name from R and S, they are unnamed in the result relation.

We illustrate $S1 \bowtie_{R.sid=S.sid} R1$ in Figure 4.13. Note that only one field called *sid* appears in the result.

sid	*sname*	*rating*	*age*	*bid*	*day*
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

Figure 4.13 $S1 \bowtie_{R.sid=S.sid} R1$

Natural Join

A further special case of the join operation $R \bowtie S$ is an equijoin in which equalities are specified on *all* fields having the same name in R and S. In this case, we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields. We call this special case a *natural join*, and it has the nice property that the result is guaranteed not to have two fields with the same name.

The equijoin expression $S1 \bowtie_{R.sid=S.sid} R1$ is actually a natural join and can simply be denoted as $S1 \bowtie R1$, since the only common field is *sid*. If the two relations have no attributes in common, $S1 \bowtie R1$ is simply the cross-product.

4.2.5 Division

The division operator is useful for expressing certain kinds of queries for example, "Find the names of sailors who have reserved all boats." Understanding how to use the basic operators of the algebra to define division is a useful exercise. However, the division operator does not have the same importance as the other operators—it is not needed as often, and database systems do not try to exploit the semantics of division by implementing it as a distinct operator (as, for example, is done with the join operator).

We discuss division through an example. Consider two relation instances A and B in which A has (exactly) two fields x and y and B has just one field y, with the same domain as in A. We define the *division* operation A/B as the set of all x values (in the form of unary tuples) such that for *every* y value in (a tuple of) B, there is a tuple $\langle x,y \rangle$ in A.

Another way to understand division is as follows. For each x value in (the first column of) A, consider the set of y values that appear in (the second field of) tuples of A with that x value. If this set contains (all y values in) B, the x value is in the result of A/B.

An analogy with integer division may also help to understand division. For integers A and B, A/B is the largest integer Q such that $Q * B \leq A$. For relation instances A and B, A/B is the largest relation instance Q such that $Q \times B \subseteq A$.

Division is illustrated in Figure 4.14. It helps to think of A as a relation listing the parts supplied by suppliers and of the B relations as listing parts. A/Bi computes suppliers who supply *all* parts listed in relation instance Bi.

Expressing A/B in terms of the basic algebra operators is an interesting exercise, and the reader should try to do this before reading further. The basic idea is to compute all x values in A that are not *disqualified*. An x value is *disqualified* if by attaching a y value from B, we obtain a tuple $\langle x,y \rangle$ that is not in A. We can compute disqualified tuples using the algebra expression

$$\pi_x((\pi_x(A) \times B) - A)$$

Thus, we can define A/B as

$$\pi_x(A) - \pi_x((\pi_x(A) \times B) - A)$$

A

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

B1

pno
p2

B2

pno
p2
p4

B3

pno
p1
p2
p4

A/B1

sno
s1
s2
s3
s4

A/B2

sno
s1
s4

A/B3

sno
s1

Figure 4.14 Examples Illustrating Division

To understand the division operation in full generality, we have to consider the case when both x and y are replaced by a set of attributes. The generalization is straightforward and left as an exercise for the reader. We discuss two additional examples illustrating division (Queries Q9 and Q10) later in this section.

4.2.6 More Examples of Algebra Queries

We now present several examples to illustrate how to write queries in relational algebra. We use the Sailors, Reserves, and Boats schema for all our examples in this section. We use parentheses as needed to make our algebra expressions unambiguous. Note that all the example queries in this chapter are given a unique query number. The query numbers are kept unique across both this chapter and the SQL query chapter (Chapter 5). This numbering makes it easy to identify a query when it is revisited in the context of relational calculus and SQL and to compare different ways of writing the same query. (All references to a query can be found in the subject index.)

In the rest of this chapter (and in Chapter 5), we illustrate queries using the instances $S3$ of Sailors, $R2$ of Reserves, and $B1$ of Boats, shown in Figures 4.15, 4.16, and 4.17, respectively.

(Q1) Find the names of sailors who have reserved boat 103.

This query can be written as follows:

$$\pi_{sname}((\sigma_{bid=103}Reserves) \bowtie Sailors)$$

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 4.15 An Instance $S3$ of Sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 4.16 An Instance $R2$ of Reserves

We first compute the set of tuples in Reserves with *bid* = 103 and then take the natural join of this set with Sailors. This expression can be evaluated on instances of Reserves and Sailors. Evaluated on the instances $R2$ and $S3$, it yields a relation that contains just one field, called *sname*, and three tuples $\langle Dustin\rangle$, $\langle Horatio\rangle$, and $\langle Lubber\rangle$. (Observe that two sailors are called Horatio and only one of them has reserved a red boat.)

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 4.17 An Instance $B1$ of Boats

We can break this query into smaller pieces using the renaming operator ρ:

$$\rho(Temp1, \sigma_{bid=103} Reserves)$$
$$\rho(Temp2, Temp1 \bowtie Sailors)$$
$$\pi_{sname}(Temp2)$$

Notice that because we are only using ρ to give names to intermediate relations, the renaming list is optional and is omitted. $Temp1$ denotes an intermediate relation that identifies reservations of boat 103. $Temp2$ is another intermediate relation, and it denotes sailors who have made a reservation in the set $Temp1$. The instances of these relations when evaluating this query on the instances $R2$ and $S3$ are illustrated in Figures 4.18 and 4.19. Finally, we extract the *sname* column from $Temp2$.

sid	*bid*	*day*
22	103	10/8/98
31	103	11/6/98
74	103	9/8/98

Figure 4.18 Instance of *Temp*1

sid	*sname*	*rating*	*age*	*bid*	*day*
22	Dustin	7	45.0	103	10/8/98
31	Lubber	8	55.5	103	11/6/98
74	Horatio	9	35.0	103	9/8/98

Figure 4.19 Instance of *Temp*2

The version of the query using ρ is essentially the same as the original query; the use of ρ is just syntactic sugar. However, there are indeed several distinct ways to write a query in relational algebra. Here is another way to write this query:

$$\pi_{sname}(\sigma_{bid=103}(Reserves \bowtie Sailors))$$

In this version we first compute the natural join of Reserves and Sailors and then apply the selection and the projection.

This example offers a glimpse of the role played by algebra in a relational DBMS. Queries are expressed by users in a language such as SQL. The DBMS translates an SQL query into (an extended form of) relational algebra and then looks for other algebra expressions that produce the same answers but are cheaper to evaluate. If the user's query is first translated into the expression

$$\pi_{sname}(\sigma_{bid=103}(Reserves \bowtie Sailors))$$

a good query optimizer will find the equivalent expression

$$\pi_{sname}((\sigma_{bid=103}Reserves) \bowtie Sailors)$$

Further, the optimizer will recognize that the second expression is likely to be less expensive to compute because the sizes of intermediate relations are smaller, thanks to the early use of selection.

(Q2) Find the names of sailors who have reserved a red boat.

$$\pi_{sname}((\sigma_{color='red'}Boats) \bowtie Reserves \bowtie Sailors)$$

This query involves a series of two joins. First, we choose (tuples describing) red boats. Then, we join this set with Reserves (natural join, with equality specified on the *bid* column) to identify reservations of red boats. Next, we join the resulting intermediate relation with Sailors (natural join, with equality specified on the *sid* column) to retrieve the names of sailors who have made reservations for red boats. Finally, we project the sailors' names. The answer, when evaluated on the instances $B1$, $R2$, and $S3$, contains the names Dustin, Horatio, and Lubber.

An equivalent expression is:

$$\pi_{sname}(\pi_{sid}((\pi_{bid}\sigma_{color='red'}Boats) \bowtie Reserves) \bowtie Sailors)$$

The reader is invited to rewrite both of these queries by using ρ to make the intermediate relations explicit and compare the schemas of the intermediate relations. The second expression generates intermediate relations with fewer fields (and is therefore likely to result in intermediate relation instances with fewer tuples as well). A relational query optimizer would try to arrive at the second expression if it is given the first.

(Q3) Find the colors of boats reserved by Lubber.

$$\pi_{color}((\sigma_{sname='Lubber'}Sailors) \bowtie Reserves \bowtie Boats)$$

This query is very similar to the query we used to compute sailors who reserved red boats. On instances $B1$, $R2$, and $S3$, the query returns the colors green and red.

(Q4) Find the names of sailors who have reserved at least one boat.

$$\pi_{sname}(Sailors \bowtie Reserves)$$

The join of Sailors and Reserves creates an intermediate relation in which tuples consist of a Sailors tuple 'attached to' a Reserves tuple. A Sailors tuple appears in (some tuple of) this intermediate relation only if at least one Reserves tuple has the same *sid* value, that is, the sailor has made some reservation. The answer, when evaluated on the instances $B1$, $R2$ and $S3$, contains the three tuples $\langle Dustin\rangle$, $\langle Horatio\rangle$, and $\langle Lubber\rangle$. Even though two sailors called Horatio have reserved a boat, the answer contains only one copy of the tuple $\langle Horatio\rangle$, because the answer is a *relation*, that is, a *set* of tuples, with no duplicates.

At this point it is worth remarking on how frequently the natural join operation is used in our examples. This frequency is more than just a coincidence based on the set of queries we have chosen to discuss; the natural join is a very natural, widely used operation. In particular, natural join is frequently used when joining two tables on a foreign key field. In Query Q4, for example, the join equates the *sid* fields of Sailors and Reserves, and the *sid* field of Reserves is a foreign key that refers to the *sid* field of Sailors.

(Q5) Find the names of sailors who have reserved a red or a green boat.

$$\rho(Tempboats, (\sigma_{color='red'}Boats) \cup (\sigma_{color='green'}Boats))$$
$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

We identify the set of all boats that are either red or green (Tempboats, which contains boats with the *bids* 102, 103, and 104 on instances $B1$, $R2$, and $S3$). Then we join with Reserves to identify *sids* of sailors who have reserved one of these boats; this gives us *sids* 22, 31, 64, and 74 over our example instances. Finally, we join (an intermediate relation containing this set of *sids*) with Sailors to find the names of Sailors with these *sids*. This gives us the names Dustin, Horatio, and Lubber on the instances $B1$, $R2$, and $S3$. Another equivalent definition is the following:

$$\rho(Tempboats, (\sigma_{color='red' \vee color='green'} Boats))$$
$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

Let us now consider a very similar query.

(Q6) Find the names of sailors who have reserved a red and a green boat. It is tempting to try to do this by simply replacing $\cup$ by $\cap$ in the definition of Tempboats:

$$\rho(Tempboats2, (\sigma_{color='red'} Boats) \cap (\sigma_{color='green'} Boats))$$
$$\pi_{sname}(Tempboats2 \bowtie Reserves \bowtie Sailors)$$

However, this solution is incorrect—it instead tries to compute sailors who have reserved a boat that is both red and green. (Since *bid* is a key for Boats, a boat can be only one color; this query will always return an empty answer set.) The correct approach is to find sailors who have reserved a red boat, then sailors who have reserved a green boat, and then take the intersection of these two sets:

$$\rho(Tempred, \pi_{sid}((\sigma_{color='red'} Boats) \bowtie Reserves))$$
$$\rho(Tempgreen, \pi_{sid}((\sigma_{color='green'} Boats) \bowtie Reserves))$$
$$\pi_{sname}((Tempred \cap Tempgreen) \bowtie Sailors)$$

The two temporary relations compute the *sids* of sailors, and their intersection identifies sailors who have reserved both red and green boats. On instances $B1$, $R2$, and $S3$, the *sids* of sailors who have reserved a red boat are 22, 31, and 64. The *sids* of sailors who have reserved a green boat are 22, 31, and 74. Thus, sailors 22 and 31 have reserved both a red boat and a green boat; their names are Dustin and Lubber.

This formulation of Query Q6 can easily be adapted to find sailors who have reserved red *or* green boats (Query Q5); just replace $\cap$ by $\cup$:

$$\rho(Tempred, \pi_{sid}((\sigma_{color='red'} Boats) \bowtie Reserves))$$
$$\rho(Tempgreen, \pi_{sid}((\sigma_{color='green'} Boats) \bowtie Reserves))$$
$$\pi_{sname}((Tempred \cup Tempgreen) \bowtie Sailors)$$

In the formulations of Queries Q5 and Q6, the fact that *sid* (the field over which we compute union or intersection) is a key for Sailors is very important. Consider the following attempt to answer Query Q6:

$$\rho(Tempred, \pi_{sname}((\sigma_{color='red'}Boats) \bowtie Reserves \bowtie Sailors))$$
$$\rho(Tempgreen, \pi_{sname}((\sigma_{color='green'}Boats) \bowtie Reserves \bowtie Sailors))$$
$$Tempred \cap Tempgreen$$

This attempt is incorrect for a rather subtle reason. Two distinct sailors with the same name, such as Horatio in our example instances, may have reserved red and green boats, respectively. In this case, the name Horatio (incorrectly) is included in the answer even though no one individual called Horatio has reserved a red boat and a green boat. The cause of this error is that *sname* is used to identify sailors (while doing the intersection) in this version of the query, but *sname* is not a key.

(Q7) Find the names of sailors who have reserved at least two boats.

$$\rho(Reservations, \pi_{sid,sname,bid}(Sailors \bowtie Reserves))$$
$$\rho(Reservationpairs(1 \rightarrow sid1, 2 \rightarrow sname1, 3 \rightarrow bid1, 4 \rightarrow sid2,$$
$$5 \rightarrow sname2, 6 \rightarrow bid2), Reservations \times Reservations)$$
$$\pi_{sname1}\sigma_{(sid1=sid2)\wedge(bid1\neq bid2)}Reservationpairs$$

First, we compute tuples of the form $\langle sid, sname, bid\rangle$, where sailor *sid* has made a reservation for boat *bid*; this set of tuples is the temporary relation Reservations. Next we find all pairs of Reservations tuples where the same sailor has made both reservations and the boats involved are distinct. Here is the central idea: To show that a sailor has reserved two boats, we must find two Reservations tuples involving the same sailor but distinct boats. Over instances $B1$, $R2$, and $S3$, each of the sailors with *sid*s 22, 31, and 64 have reserved at least two boats. Finally, we project the names of such sailors to obtain the answer, containing the names Dustin, Horatio, and Lubber.

Notice that we included *sid* in Reservations because it is the key field identifying sailors, and we need it to check that two Reservations tuples involve the same sailor. As noted in the previous example, we cannot use *sname* for this purpose.

(Q8) Find the sids of sailors with age over 20 who have not reserved a red boat.

$$\pi_{sid}(\sigma_{age>20}Sailors) -$$
$$\pi_{sid}((\sigma_{color='red'}Boats) \bowtie Reserves \bowtie Sailors)$$

This query illustrates the use of the set-difference operator. Again, we use the fact that *sid* is the key for Sailors. We first identify sailors aged over 20 (over instances $B1$, $R2$, and $S3$, *sids* 22, 29, 31, 32, 58, 64, 74, 85, and 95) and then discard those who have reserved a red boat (*sids* 22, 31, and 64), to obtain the answer (*sids* 29, 32, 58, 74, 85, and 95). If we want to compute the names of such sailors, we must first compute their *sids* (as shown earlier) and then join with Sailors and project the *sname* values.

(Q9) Find the names of sailors who have reserved all boats.

The use of the word *all* (or *every*) is a good indication that the division operation might be applicable:

$$\rho(Tempsids, (\pi_{sid,bid}Reserves)/(\pi_{bid}Boats))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

The intermediate relation Tempsids is defined using division and computes the set of *sids* of sailors who have reserved every boat (over instances $B1$, $R2$, and $S3$, this is just *sid* 22). Note how we define the two relations that the division operator (/) is applied to—the first relation has the schema *(sid,bid)* and the second has the schema *(bid)*. Division then returns all *sids* such that there is a tuple $\langle sid, bid \rangle$ in the first relation for each *bid* in the second. Joining Tempsids with Sailors is necessary to associate names with the selected *sids*; for sailor 22, the name is Dustin.

(Q10) Find the names of sailors who have reserved all boats called Interlake.

$$\rho(Tempsids, (\pi_{sid,bid}Reserves)/(\pi_{bid}(\sigma_{bname='Interlake'}Boats)))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

The only difference with respect to the previous query is that now we apply a selection to Boats, to ensure that we compute *bids* only of boats named *Interlake* in defining the second argument to the division operator. Over instances $B1$, $R2$, and $S3$, Tempsids evaluates to *sids* 22 and 64, and the answer contains their names, Dustin and Horatio.

4.3 RELATIONAL CALCULUS

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or *declarative*, in that it allows us to describe the set of answers without being explicit about how they should be computed. Relational calculus has had a big influence on the design of commercial query languages such as SQL and, especially, Query-by-Example (QBE).

The variant of the calculus we present in detail is called the **tuple relational calculus (TRC)**. Variables in TRC take on tuples as values. In another variant, called the **domain relational calculus (DRC)**, the variables range over field values. TRC has had more of an influence on SQL, while DRC has strongly influenced QBE. We discuss DRC in Section 4.3.2.[2]

4.3.1 Tuple Relational Calculus

A **tuple variable** is a variable that takes on tuples of a particular relation schema as values. That is, every value assigned to a given tuple variable has the same number and type of fields. A tuple relational calculus query has the form $\{ T \mid p(T) \}$, where T is a tuple variable and $p(T)$ denotes a *formula* that describes T; we will shortly define formulas and queries rigorously. The result of this query is the set of all tuples t for which the formula $p(T)$ evaluates to `true` with $T = t$. The language for writing formulas $p(T)$ is thus at the heart of TRC and essentially a simple subset of *first-order logic*. As a simple example, consider the following query.

(Q11) Find all sailors with a rating above 7.

$$\{S \mid S \in Sailors \wedge S.rating > 7\}$$

When this query is evaluated on an instance of the Sailors relation, the tuple variable S is instantiated successively with each tuple, and the test *S.rating*>7 is applied. The answer contains those instances of S that pass this test. On instance $S3$ of Sailors, the answer contains Sailors tuples with *sid* 31, 32, 58, 71, and 74.

Syntax of TRC Queries

We now define these concepts formally, beginning with the notion of a formula. Let *Rel* be a relation name, R and S be tuple variables, a be an attribute of R, and b be an attribute of S. Let op denote an operator in the set $\{<, >, =, \leq, \geq, \neq\}$. An **atomic formula** is one of the following:

- $R \in Rel$
- $R.a$ op $S.b$
- $R.a$ op *constant*, or *constant* op $R.a$

A **formula** is recursively defined to be one of the following, where p and q are themselves formulas and $p(R)$ denotes a formula in which the variable R appears:

[2]The material on DRC is referred to in the (online) chapter on QBE; with the exception of this chapter, the material on DRC and TRC can be omitted without loss of continuity.

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists R(p(R))$, where R is a tuple variable
- $\forall R(p(R))$, where R is a tuple variable

In the last two clauses, the **quantifiers** $\exists$ and $\forall$ are said to **bind** the variable R. A variable is said to be **free** in a formula or *subformula* (a formula contained in a larger formula) if the (sub)formula does not contain an occurrence of a quantifier that binds it.[3]

We observe that every variable in a TRC formula appears in a subformula that is atomic, and every relation schema specifies a domain for each field; this observation ensures that each variable in a TRC formula has a well-defined domain from which values for the variable are drawn. That is, each variable has a well-defined *type*, in the programming language sense. Informally, an atomic formula $R \in Rel$ gives R the type of tuples in *Rel*, and comparisons such as $R.a$ `op` $S.b$ and $R.a$ `op` *constant* induce type restrictions on the field $R.a$. If a variable R does not appear in an atomic formula of the form $R \in Rel$ (i.e., it appears only in atomic formulas that are comparisons), we follow the convention that the type of R is a tuple whose fields include all (and only) fields of R that appear in the formula.

We do not define types of variables formally, but the type of a variable should be clear in most cases, and the important point to note is that comparisons of values having different types should always fail. (In discussions of relational calculus, the simplifying assumption is often made that there is a single domain of constants and this is the domain associated with each field of each relation.)

A **TRC query** is defined to be expression of the form $\{T \mid p(T)\}$, where T is the only free variable in the formula p.

Semantics of TRC Queries

What does a TRC query mean? More precisely, what is the set of answer tuples for a given TRC query? The **answer** to a TRC query $\{T \mid p(T)\}$, as noted earlier, is the set of all tuples t for which the formula $p(T)$ evaluates to `true` with variable T assigned the tuple value t. To complete this definition, we must state which assignments of tuple values to the free variables in a formula make the formula evaluate to `true`.

[3]We make the assumption that each variable in a formula is either free or bound by exactly one occurrence of a quantifier, to avoid worrying about details such as nested occurrences of quantifiers that bind some, but not all, occurrences of variables.

A query is evaluated on a given instance of the database. Let each free variable in a formula F be bound to a tuple value. For the given assignment of tuples to variables, with respect to the given database instance, F evaluates to (or simply 'is') `true` if one of the following holds:

- F is an atomic formula $R \in Rel$, and R is assigned a tuple in the instance of relation Rel.
- F is a comparison $R.a$ `op` $S.b$, $R.a$ `op` *constant*, or *constant* `op` $R.a$, and the tuples assigned to R and S have field values $R.a$ and $S.b$ that make the comparison `true`.
- F is of the form $\neg p$ and p is not `true`, or of the form $p \wedge q$, and both p and q are `true`, or of the form $p \vee q$ and one of them is `true`, or of the form $p \Rightarrow q$ and q is `true` whenever[4] p is `true`.
- F is of the form $\exists R(p(R))$, and there is some assignment of tuples to the free variables in $p(R)$, including the variable R,[5] that makes the formula $p(R)$ `true`.
- F is of the form $\forall R(p(R))$, and there is some assignment of tuples to the free variables in $p(R)$ that makes the formula $p(R)$ `true` no matter what tuple is assigned to R.

Examples of TRC Queries

We now illustrate the calculus through several examples, using the instances $B1$ of Boats, $R2$ of Reserves, and $S3$ of Sailors shown in Figures 4.15, 4.16, and 4.17. We use parentheses as needed to make our formulas unambiguous. Often, a formula $p(R)$ includes a condition $R \in Rel$, and the meaning of the phrases *some tuple* R and *for all tuples* R is intuitive. We use the notation $\exists R \in Rel(p(R))$ for $\exists R(R \in Rel \wedge p(R))$. Similarly, we use the notation $\forall R \in Rel(p(R))$ for $\forall R(R \in Rel \Rightarrow p(R))$.

(Q12) Find the names and ages of sailors with a rating above 7.

$$\{P \mid \exists S \in Sailors(S.rating > 7 \wedge P.name = S.sname \wedge P.age = S.age)\}$$

This query illustrates a useful convention: P is considered to be a tuple variable with exactly two fields, which are called *name* and *age*, because these are the only fields of P mentioned and P does not range over any of the relations in the query; that is, there is no subformula of the form $P \in Relname$. The result of this query is a relation with two fields, *name* and *age*. The atomic

[4] *Whenever* should be read more precisely as 'for all assignments of tuples to the free variables.'
[5] Note that some of the free variables in $p(R)$ (e.g., the variable R itself) may be bound in F.

formulas $P.name = S.sname$ and $P.age = S.age$ give values to the fields of an answer tuple P. On instances $B1$, $R2$, and $S3$, the answer is the set of tuples $\langle Lubber, 55.5\rangle$, $\langle Andy, 25.5\rangle$, $\langle Rusty, 35.0\rangle$, $\langle Zorba, 16.0\rangle$, and $\langle Horatio, 35.0\rangle$.

(Q13) Find the sailor name, boat id, and reservation date for each reservation.

$$\{P \mid \exists R \in Reserves \;\; \exists S \in Sailors$$
$$(R.sid = S.sid \wedge P.bid = R.bid \wedge P.day = R.day \wedge P.sname = S.sname)\}$$

For each Reserves tuple, we look for a tuple in Sailors with the same *sid*. Given a pair of such tuples, we construct an answer tuple P with fields *sname, bid*, and *day* by copying the corresponding fields from these two tuples. This query illustrates how we can combine values from different relations in each answer tuple. The answer to this query on instances $B1$, $R2$, and $S3$ is shown in Figure 4.20.

sname	*bid*	*day*
Dustin	101	10/10/98
Dustin	102	10/10/98
Dustin	103	10/8/98
Dustin	104	10/7/98
Lubber	102	11/10/98
Lubber	103	11/6/98
Lubber	104	11/12/98
Horatio	101	9/5/98
Horatio	102	9/8/98
Horatio	103	9/8/98

Figure 4.20 Answer to Query Q13

(Q1) Find the names of sailors who have reserved boat 103.

$$\{P \mid \exists S \in Sailors \;\; \exists R \in Reserves(R.sid = S.sid \wedge R.bid = 103$$
$$\wedge P.sname = S.sname)\}$$

This query can be read as follows: "Retrieve all sailor tuples for which there exists a tuple in Reserves having the same value in the *sid* field and with $bid = 103$." That is, for each sailor tuple, we look for a tuple in Reserves that shows that this sailor has reserved boat 103. The answer tuple P contains just one field, *sname*.

(Q2) Find the names of sailors who have reserved a red boat.

$$\{P \mid \exists S \in Sailors \;\; \exists R \in Reserves(R.sid = S.sid \wedge P.sname = S.sname$$

$$\wedge \exists B \in Boats(B.bid = R.bid \wedge B.color = 'red'))\}$$

This query can be read as follows: "Retrieve all sailor tuples S for which there exist tuples R in Reserves and B in Boats such that $S.sid = R.sid$, $R.bid = B.bid$, and $B.color = 'red'$." Another way to write this query, which corresponds more closely to this reading, is as follows:

$$\{P \mid \exists S \in Sailors \;\; \exists R \in Reserves \;\; \exists B \in Boats$$
$$(R.sid = S.sid \wedge B.bid = R.bid \wedge B.color = 'red' \wedge P.sname = S.sname)\}$$

(Q7) Find the names of sailors who have reserved at least two boats.

$$\{P \mid \exists S \in Sailors \;\; \exists R1 \in Reserves \;\; \exists R2 \in Reserves$$
$$(S.sid = R1.sid \wedge R1.sid = R2.sid \wedge R1.bid \neq R2.bid$$
$$\wedge P.sname = S.sname)\}$$

Contrast this query with the algebra version and see how much simpler the calculus version is. In part, this difference is due to the cumbersome renaming of fields in the algebra version, but the calculus version really is simpler.

(Q9) Find the names of sailors who have reserved all boats.

$$\{P \mid \exists S \in Sailors \;\; \forall B \in Boats$$
$$(\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid \wedge P.sname = S.sname))\}$$

This query was expressed using the division operator in relational algebra. Note how easily it is expressed in the calculus. The calculus query directly reflects how we might express the query in English: "Find sailors S such that for all boats B there is a Reserves tuple showing that sailor S has reserved boat B."

(Q14) Find sailors who have reserved all red boats.

$$\{S \mid S \in Sailors \wedge \forall B \in Boats$$
$$(B.color = 'red' \Rightarrow (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query can be read as follows: For each candidate (sailor), if a boat is red, the sailor must have reserved it. That is, for a candidate sailor, a boat being red must imply that the sailor has reserved it. Observe that since we can return an entire sailor tuple as the answer instead of just the sailor's name, we avoided introducing a new free variable (e.g., the variable P in the previous example) to hold the answer values. On instances $B1$, $R2$, and $S3$, the answer contains the Sailors tuples with *sids* 22 and 31.

We can write this query without using implication, by observing that an expression of the form $p \Rightarrow q$ is logically equivalent to $\neg p \vee q$:

$$\{S \mid S \in Sailors \wedge \forall B \in Boats$$

$$(B.color \neq' red' \vee (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query should be read as follows: "Find sailors S such that, for all boats B, either the boat is not red or a Reserves tuple shows that sailor S has reserved boat B."

4.3.2 Domain Relational Calculus

A **domain variable** is a variable that ranges over the values in the domain of some attribute (e.g., the variable can be assigned an integer if it appears in an attribute whose domain is the set of integers). A DRC query has the form $\{\langle x_1, x_2, \ldots, x_n \rangle \mid p(\langle x_1, x_2, \ldots, x_n \rangle)\}$, where each x_i is either a *domain variable* or a constant and $p(\langle x_1, x_2, \ldots, x_n \rangle)$ denotes a **DRC formula** whose only free variables are the variables among the x_i, $1 \leq i \leq n$. The result of this query is the set of all tuples $\langle x_1, x_2, \ldots, x_n \rangle$ for which the formula evaluates to `true`.

A DRC formula is defined in a manner very similar to the definition of a TRC formula. The main difference is that the variables are now domain variables. Let `op` denote an operator in the set $\{<, >, =, \leq, \geq, \neq\}$ and let X and Y be domain variables. An **atomic formula** in DRC is one of the following:

- $\langle x_1, x_2, \ldots, x_n \rangle \in Rel$, where Rel is a relation with n attributes; each x_i, $1 \leq i \leq n$ is either a variable or a constant
- X `op` Y
- X `op` *constant*, or *constant* `op` X

A **formula** is recursively defined to be one of the following, where p and q are themselves formulas and $p(X)$ denotes a formula in which the variable X appears:

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists X(p(X))$, where X is a domain variable
- $\forall X(p(X))$, where X is a domain variable

The reader is invited to compare this definition with the definition of TRC formulas and see how closely these two definitions correspond. We will not define the semantics of DRC formulas formally; this is left as an exercise for the reader.

Examples of DRC Queries

We now illustrate DRC through several examples. The reader is invited to compare these with the TRC versions.

(Q11) Find all sailors with a rating above 7.

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in \mathit{Sailors} \wedge T > 7\}$$

This differs from the TRC version in giving each attribute a (variable) name. The condition $\langle I, N, T, A\rangle \in \mathit{Sailors}$ ensures that the domain variables I, N, T, and A are restricted to be fields of the *same* tuple. In comparison with the TRC query, we can say $T > 7$ instead of $S.\mathit{rating} > 7$, but we must specify the tuple $\langle I, N, T, A\rangle$ in the result, rather than just S.

(Q1) Find the names of sailors who have reserved boat 103.

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in \mathit{Sailors} \\ \wedge \exists Ir, Br, D(\langle Ir, Br, D\rangle \in \mathit{Reserves} \wedge Ir = I \wedge Br = 103))\}$$

Note that only the *sname* field is retained in the answer and that only N is a free variable. We use the notation $\exists Ir, Br, D(\ldots)$ as a shorthand for $\exists Ir(\exists Br(\exists D(\ldots)))$. Very often, all the quantified variables appear in a single relation, as in this example. An even more compact notation in this case is $\exists\langle Ir, Br, D\rangle \in \mathit{Reserves}$. With this notation, which we use henceforth, the query would be as follows:

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in \mathit{Sailors} \\ \wedge \exists\langle Ir, Br, D\rangle \in \mathit{Reserves}(Ir = I \wedge Br = 103))\}$$

The comparison with the corresponding TRC formula should now be straightforward. This query can also be written as follows; note the repetition of variable I and the use of the constant 103:

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in \mathit{Sailors} \\ \wedge \exists D(\langle I, 103, D\rangle \in \mathit{Reserves}))\}$$

(Q2) Find the names of sailors who have reserved a red boat.

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in \mathit{Sailors} \\ \wedge \exists\langle I, Br, D\rangle \in \mathit{Reserves} \wedge \exists\langle Br, BN, '\mathit{red}'\rangle \in \mathit{Boats})\}$$

(Q7) Find the names of sailors who have reserved at least two boats.

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in \mathit{Sailors} \wedge \\ \exists Br1, Br2, D1, D2(\langle I, Br1, D1\rangle \in \mathit{Reserves} \\ \wedge \langle I, Br2, D2\rangle \in \mathit{Reserves} \wedge Br1 \neq Br2))\}$$

Note how the repeated use of variable I ensures that the same sailor has reserved both the boats in question.

(Q9) Find the names of sailors who have reserved all boats.

$$\begin{aligned}&\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in \mathit{Sailors} \wedge \\ &\forall B, BN, C(\neg(\langle B, BN, C\rangle \in \mathit{Boats}) \vee \\ &(\exists\langle Ir, Br, D\rangle \in \mathit{Reserves}(I = Ir \wedge Br = B))))\}\end{aligned}$$

This query can be read as follows: "Find all values of N such that some tuple $\langle I, N, T, A\rangle$ in Sailors satisfies the following condition: For every $\langle B, BN, C\rangle$, either this is not a tuple in Boats or there is some tuple $\langle Ir, Br, D\rangle$ in Reserves that proves that Sailor I has reserved boat B." The $\forall$ quantifier allows the domain variables B, BN, and C to range over all values in their respective attribute domains, and the pattern '$\neg(\langle B, BN, C\rangle \in \mathit{Boats})\vee$' is necessary to restrict attention to those values that appear in tuples of Boats. This pattern is common in DRC formulas, and the notation $\forall\langle B, BN, C\rangle \in \mathit{Boats}$ can be used as a shortcut instead. This is similar to the notation introduced earlier for $\exists$. With this notation, the query would be written as follows:

$$\begin{aligned}&\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in \mathit{Sailors} \wedge \forall\langle B, BN, C\rangle \in \mathit{Boats} \\ &(\exists\langle Ir, Br, D\rangle \in \mathit{Reserves}(I = Ir \wedge Br = B)))\}\end{aligned}$$

(Q14) Find sailors who have reserved all red boats.

$$\begin{aligned}&\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in \mathit{Sailors} \wedge \forall\langle B, BN, C\rangle \in \mathit{Boats} \\ &(C = 'red' \Rightarrow \exists\langle Ir, Br, D\rangle \in \mathit{Reserves}(I = Ir \wedge Br = B))\}\end{aligned}$$

Here, we find all sailors such that, for every red boat, there is a tuple in Reserves that shows the sailor has reserved it.

4.4 EXPRESSIVE POWER OF ALGEBRA AND CALCULUS

We presented two formal query languages for the relational model. Are they equivalent in power? Can every query that can be expressed in relational algebra also be expressed in relational calculus? The answer is yes, it can. Can every query that can be expressed in relational calculus also be expressed in relational algebra? Before we answer this question, we consider a major problem with the calculus as we presented it.

Consider the query $\{S \mid \neg(S \in \mathit{Sailors})\}$. This query is syntactically correct. However, it asks for all tuples S such that S is not in (the given instance of)

Sailors. The set of such S tuples is obviously infinite, in the context of infinite domains such as the set of all integers. This simple example illustrates an *unsafe* query. It is desirable to restrict relational calculus to disallow unsafe queries.

We now sketch how calculus queries are restricted to be safe. Consider a set I of relation instances, with one instance per relation that appears in the query Q. Let $Dom(Q, I)$ be the set of all constants that appear in these relation instances I or in the formulation of the query Q itself. Since we allow only finite instances I, $Dom(Q, I)$ is also finite.

For a calculus formula Q to be considered safe, at a minimum we want to ensure that, for any given I, the set of answers for Q contains only values in $Dom(Q, I)$. While this restriction is obviously required, it is not enough. Not only do we want the set of answers to be composed of constants in $Dom(Q, I)$, we wish to *compute* the set of answers by examining only tuples that contain constants in $Dom(Q, I)$! This wish leads to a subtle point associated with the use of quantifiers $\forall$ and $\exists$: Given a TRC formula of the form $\exists R(p(R))$, we want to find all values for variable R that make this formula `true` by checking only tuples that contain constants in $Dom(Q, I)$. Similarly, given a TRC formula of the form $\forall R(p(R))$, we want to find any values for variable R that make this formula `false` by checking only tuples that contain constants in $Dom(Q, I)$.

We therefore define a *safe* TRC formula Q to be a formula such that:

1. For any given I, the set of answers for Q contains only values that are in $Dom(Q, I)$.
2. For each subexpression of the form $\exists R(p(R))$ in Q, if a tuple r (assigned to variable R) makes the formula `true`, then r contains only constants in $Dom(Q, I)$.
3. For each subexpression of the form $\forall R(p(R))$ in Q, if a tuple r (assigned to variable R) contains a constant that is not in $Dom(Q, I)$, then r must make the formula `true`.

Note that this definition is not *constructive*, that is, it does not tell us how to check if a query is safe.

The query $Q = \{S \mid \neg(S \in Sailors)\}$ is unsafe by this definition. $Dom(Q, I)$ is the set of all values that appear in (an instance I of) Sailors. Consider the instance $S1$ shown in Figure 4.1. The answer to this query obviously includes values that do not appear in $Dom(Q, S1)$.

Returning to the question of expressiveness, we can show that every query that can be expressed using a *safe* relational calculus query can also be expressed as a relational algebra query. The expressive power of relational algebra is often used as a metric of how powerful a relational database query language is. If a query language can express all the queries that we can express in relational algebra, it is said to be **relationally complete**. A practical query language is expected to be relationally complete; in addition, commercial query languages typically support features that allow us to express some queries that cannot be expressed in relational algebra.

4.5 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What is the input to a relational query? What is the result of evaluating a query? **(Section 4.1)**
- Database systems use some variant of relational algebra to represent query evaluation plans. Explain why algebra is suitable for this purpose. **(Section 4.2)**
- Describe the selection operator. What can you say about the cardinality of the input and output tables for this operator? (That is, if the input has k tuples, what can you say about the output?) Describe the projection operator. What can you say about the cardinality of the input and output tables for this operator? **(Section 4.2.1)**
- Describe the set operations of relational algebra, including union ($\cup$), intersection ($\cap$), set-difference ($-$), and cross-product ($\times$). For each, what can you say about the cardinality of their input and output tables? **(Section 4.2.2)**
- Explain how the renaming operator is used. Is it required? That is, if this operator is not allowed, is there any query that can no longer be expressed in algebra? **(Section 4.2.3)**
- Define all the variations of the join operation. Why is the join operation given special attention? Cannot we express every join operation in terms of cross-product, selection, and projection? **(Section 4.2.4)**
- Define the division operation in terms of the basic relational algebra operations. Describe a typical query that calls for division. Unlike join, the division operator is not given special treatment in database systems. Explain why. **(Section 4.2.5)**

- Relational calculus is said to be a *declarative* language, in contrast to algebra, which is a *procedural* language. Explain the distinction. **(Section 4.3)**
- How does a relational calculus query 'describe' result tuples? Discuss the subset of first-order predicate logic used in tuple relational calculus, with particular attention to universal and existential quantifiers, bound and free variables, and restrictions on the query formula. **(Section 4.3.1)**.
- What is the difference between tuple relational calculus and domain relational calculus? **(Section 4.3.2)**.
- What is an *unsafe* calculus query? Why is it important to avoid such queries? **(Section 4.4)**
- Relational algebra and relational calculus are said to be equivalent in expressive power. Explain what this means, and how it is related to the notion of *relational completeness.* **(Section 4.4)**

EXERCISES

Exercise 4.1 Explain the statement that relational algebra operators can be *composed.* Why is the ability to compose operators important?

Exercise 4.2 Given two relations $R1$ and $R2$, where $R1$ contains N1 tuples, $R2$ contains N2 tuples, and N2 > N1 > 0, give the minimum and maximum possible sizes (in tuples) for the resulting relation produced by each of the following relational algebra expressions. In each case, state any assumptions about the schemas for $R1$ and $R2$ needed to make the expression meaningful:

(1) $R1 \cup R2$, (2) $R1 \cap R2$, (3) $R1 - R2$, (4) $R1 \times R2$, (5) $\sigma_{a=5}(R1)$, (6) $\pi_a(R1)$, and (7) $R1/R2$

Exercise 4.3 Consider the following schema:

Suppliers(<u>*sid:* `integer`</u>, *sname:* `string`, *address:* `string`)
Parts(<u>*pid:* `integer`</u>, *pname:* `string`, *color:* `string`)
Catalog(<u>*sid:* `integer`, *pid:* `integer`</u>, *cost:* `real`)

The key fields are underlined, and the domain of each field is listed after the field name. Therefore *sid* is the key for Suppliers, *pid* is the key for Parts, and *sid* and *pid* together form the key for Catalog. The Catalog relation lists the prices charged for parts by Suppliers. Write the following queries in relational algebra, tuple relational calculus, and domain relational calculus:

1. Find the *names* of suppliers who supply some red part.
2. Find the *sids* of suppliers who supply some red or green part.
3. Find the *sids* of suppliers who supply some red part or are at 221 Packer Ave.
4. Find the *sids* of suppliers who supply some red part and some green part.

5. Find the *sids* of suppliers who supply every part.
6. Find the *sids* of suppliers who supply every red part.
7. Find the *sids* of suppliers who supply every red or green part.
8. Find the *sids* of suppliers who supply every red part or supply every green part.
9. Find pairs of *sids* such that the supplier with the first *sid* charges more for some part than the supplier with the second *sid*.
10. Find the *pids* of parts supplied by at least two different suppliers.
11. Find the *pids* of the most expensive parts supplied by suppliers named Yosemite Sham.
12. Find the *pids* of parts supplied by every supplier at less than \$200. (If any supplier either does not supply the part or charges more than \$200 for it, the part is not selected.)

Exercise 4.4 Consider the Supplier-Parts-Catalog schema from the previous question. State what the following queries compute:

1. $\pi_{sname}(\pi_{sid}(\sigma_{color='red'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers)$
2. $\pi_{sname}(\pi_{sid}((\sigma_{color='red'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers))$
3. $(\pi_{sname}((\sigma_{color='red'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers)) \cap$
$$(\pi_{sname}((\sigma_{color='green'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers))$$
4. $(\pi_{sid}((\sigma_{color='red'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers)) \cap$
$$(\pi_{sid}((\sigma_{color='green'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers))$$
5. $\pi_{sname}((\pi_{sid,sname}((\sigma_{color='red'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers)) \cap$
$$(\pi_{sid,sname}((\sigma_{color='green'}Parts) \bowtie (\sigma_{cost<100}Catalog) \bowtie Suppliers)))$$

Exercise 4.5 Consider the following relations containing airline flight information:

Flights(*flno:* `integer`, *from:* `string`, *to:* `string`,
distance: `integer`, *departs:* `time`, *arrives:* `time`)
Aircraft(*aid:* `integer`, *aname:* `string`, *cruisingrange:* `integer`)
Certified(*eid:* `integer`, *aid:* `integer`)
Employees(*eid:* `integer`, *ename:* `string`, *salary:* `integer`)

Note that the Employees relation describes pilots and other kinds of employees as well; every pilot is certified for some aircraft (otherwise, he or she would not qualify as a pilot), and only pilots are certified to fly.

Write the following queries in relational algebra, tuple relational calculus, and domain relational calculus. Note that some of these queries may not be expressible in relational algebra (and, therefore, also not expressible in tuple and domain relational calculus)! For such queries, informally explain why they cannot be expressed. (See the exercises at the end of Chapter 5 for additional queries over the airline schema.)

1. Find the *eids* of pilots certified for some Boeing aircraft.
2. Find the *names* of pilots certified for some Boeing aircraft.
3. Find the *aids* of all aircraft that can be used on non-stop flights from Bonn to Madras.

4. Identify the flights that can be piloted by every pilot whose salary is more than $100,000.
5. Find the names of pilots who can operate planes with a range greater than 3,000 miles but are not certified on any Boeing aircraft.
6. Find the *eids* of employees who make the highest salary.
7. Find the *eids* of employees who make the second highest salary.
8. Find the *eids* of employees who are certified for the largest number of aircraft.
9. Find the *eids* of employees who are certified for exactly three aircraft.
10. Find the total amount paid to employees as salaries.
11. Is there a sequence of flights from Madison to Timbuktu? Each flight in the sequence is required to depart from the city that is the destination of the previous flight; the first flight must leave Madison, the last flight must reach Timbuktu, and there is no restriction on the number of intermediate flights. Your query must determine whether a sequence of flights from Madison to Timbuktu exists for *any* input Flights relation instance.

Exercise 4.6 What is *relational completeness*? If a query language is relationally complete, can you write any desired query in that language?

Exercise 4.7 What is an *unsafe* query? Give an example and explain why it is important to disallow such queries.

BIBLIOGRAPHIC NOTES

Relational algebra was proposed by Codd in [187], and he showed the equivalence of relational algebra and TRC in [189]. Earlier, Kuhns [454] considered the use of logic to pose queries. LaCroix and Pirotte discussed DRC in [459]. Klug generalized the algebra and calculus to include aggregate operations in [439]. Extensions of the algebra and calculus to deal with aggregate functions are also discussed in [578]. Merrett proposed an extended relational algebra with quantifiers such as *the number of* that go beyond just universal and existential quantification [530]. Such generalized quantifiers are discussed at length in [52].