Chapter 6

The Database Language SQL

The most commonly used relational DBMS's query and modify the database through a language called SQL (sometimes pronounced "sequel"). SQL stands for "Structured Query Language." The portion of SQL that supports queries has capabilities very close to that of relational algebra, as extended in Section 5.2. However, SQL also includes statements for modifying the database (e.g., inserting and deleting tuples from relations) and for declaring a database schema. Thus, SQL serves as both a data-manipulation language and as a data-definition language. SQL also standardizes many other database commands, covered in Chapters 7 and 9.

There are many different dialects of SQL. First, there are three major standards. There is ANSI (American National Standards Institute) SQL and an updated standard adopted in 1992, called SQL-92 or SQL2. The most recent SQL-99 (previously referred to as SQL3) standard extends SQL2 with object-relational features and a number of other new capabilities. There is also a collection of extensions to SQL-99, collectively called SQL:2003. Then, there are versions of SQL produced by the principal DBMS vendors. These all include the capabilities of the original ANSI standard. They also conform to a large extent to the more recent SQL2, although each has its variations and extensions beyond SQL2, including some, but not all, of the features in the SQL-99 and SQL:2003 standards.

This chapter introduces the basics of SQL: the query language and database modification statements. We also introduce the notion of a "transaction," the basic unit of work for database systems. This study, although simplifed, will give you a sense of how database operations can interact and some of the resulting pitfalls.

The next chapter discusses constraints and triggers, as another way of exerting user control over the content of the database. Chapter 8 covers some of the ways that we can make our SQL queries more efficient, principally by declaring indexes and related structures. Chapter 9 covers database-related programming as part of a whole system, such as the servers that we commonly access over the Web. There, we shall see that SQL queries and other operations are almost never performed in isolation, but are embedded in a conventional host language, with which it must interact.

Finally, Chapter 10 explains a number of advanced database programming concepts. These include recursive SQL, security and access control in SQL, object-relational SQL, and the data-cube model of data.

The intent of this chapter and the following are to provide the reader with a sense of what SQL is about, more at the level of a "tutorial" than a "manual." Thus, we focus on the most commonly used features only, and we try to use code that not only conforms to the standard, but to the usage of commercial DBMS's. The references mention places where more of the details of the language and its dialects can be found.

6.1 Simple Queries in SQL

Perhaps the simplest form of query in SQL asks for those tuples of some one relation that satisfy a condition. Such a query is analogous to a selection in relational algebra. This simple query, like almost all SQL queries, uses the three keywords, `SELECT`, `FROM`, and `WHERE` that characterize SQL.

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Figure 6.1: Example database schema, repeated

Example 6.1: In this and subsequent examples, we shall use the movie database schema from Section 2.2.8. For reference, these relation schemas are the ones shown in Fig. 6.1.

As our first query, let us ask about the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

for all movies produced by Disney Studios in 1990. In SQL, we say

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

This query exhibits the characteristic select-from-where form of most SQL queries.

How SQL is Used

In this chapter, we assume a *generic query interface*, where we type SQL queries or other statements and have them execute. In practice, the generic interface is used rarely. Rather, there are large programs, written in a conventional language such as C or Java (called the *host language*). These programs issue SQL statements to a database, using a special library for the host language. Data is moved from host-language variables to the SQL statements, and the results of those statements are moved from the database to host-language variables. We shall have more to say about the matter in Chapter 9.

- The `FROM` clause gives the relation or relations to which the query refers. In our example, the query is about the relation `Movies`.
- The `WHERE` clause is a condition, much like a selection-condition in relational algebra. Tuples must satisfy the condition in order to match the query. Here, the condition is that the `studioName` attribute of the tuple has the value `'Disney'` and the `year` attribute of the tuple has the value 1990. All tuples meeting both stipulations satisfy the condition; other tuples do not.
- The `SELECT` clause tells which attributes of the tuples matching the condition are produced as part of the answer. The `*` in this example indicates that the entire tuple is produced. The result of the query is the relation consisting of all tuples produced by this process.

One way to interpret this query is to consider each tuple of the relation mentioned in the `FROM` clause. The condition in the `WHERE` clause is applied to the tuple. More precisely, any attributes mentioned in the `WHERE` clause are replaced by the value in the tuple's component for that attribute. The condition is then evaluated, and if true, the components appearing in the `SELECT` clause are produced as one tuple of the answer. Thus, the result of the query is the `Movies` tuples for those movies produced by Disney in 1990, for example, *Pretty Woman*.

In detail, when the SQL query processor encounters the `Movies` tuple

title	*year*	*length*	*genre*	*studioName*	*producerC#*
`Pretty Woman`	`1990`	`119`	`romance`	`Disney`	`999`

(here, 999 is the imaginary certificate number for the producer of the movie), the value `'Disney'` is substituted for attribute `studioName` and value 1990 is substituted for attribute `year` in the condition of the `WHERE` clause, because these are the values for those attributes in the tuple in question. The `WHERE` clause thus becomes

A Trick for Reading and Writing Queries

It is generally easist to examine a select-from-where query by first looking at the FROM clause, to learn which relations are involved in the query. Then, move to the WHERE clause, to learn what it is about tuples that is important to the query. Finally, look at the SELECT clause to see what the output is. The same order — from, then where, then select — is often useful when writing queries of your own, as well.

```
WHERE 'Disney' = 'Disney' AND 1990 = 1990
```

Since this condition is evidently true, the tuple for *Pretty Woman* passes the test of the WHERE clause and the tuple becomes part of the result of the query. □

6.1.1 Projection in SQL

We can, if we wish, eliminate some of the components of the chosen tuples; that is, we can project the relation produced by a SQL query onto some of its attributes. In place of the * of the SELECT clause, we may list some of the attributes of the relation mentioned in the FROM clause. The result will be projected onto the attributes listed.[1]

Example 6.2: Suppose we wish to modify the query of Example 6.1 to produce only the movie title and length. We may write

```
SELECT title, length
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

The result is a table with two columns, headed `title` and `length`. The tuples in this table are pairs, each consisting of a movie title and its length, such that the movie was produced by Disney in 1990. For instance, the relation schema and one of its tuples looks like:

title	*length*
Pretty Woman	119
...	...

□

[1]Thus, the keyword SELECT in SQL actually corresponds most closely to the projection operator of relational algebra, while the selection operator of the algebra corresponds to the WHERE clause of SQL queries.

Sometimes, we wish to produce a relation with column headers different from the attributes of the relation mentioned in the FROM clause. We may follow the name of the attribute by the keyword AS and an *alias*, which becomes the header in the result relation. Keyword AS is optional. That is, an alias can immediately follow what it stands for, without any intervening punctuation.

Example 6.3: We can modify Example 6.2 to produce a relation with attributes name and duration in place of title and length as follows.

```
SELECT title AS name, length AS duration
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

The result is the same set of tuples as in Example 6.2, but with the columns headed by attributes name and duration. For example,

name	*duration*
Pretty Woman	119
...	...

could be the first tuple in the result. □

Another option in the SELECT clause is to use an expression in place of an attribute. Put another way, the SELECT list can function like the lists in an extended projection, which we discussed in Section 5.2.5. We shall see in Section 6.4 that the SELECT list can also include aggregates as in the γ operator of Section 5.2.4.

Example 6.4: Suppose we want output as in Example 6.3, but with the length in hours. We might replace the SELECT clause of that example with

```
SELECT title AS name, length*0.016667 AS lengthInHours
```

Then the same movies would be produced, but lengths would be calculated in hours and the second column would be headed by attribute lengthInHours, as:

name	*lengthInHours*
Pretty Woman	1.98334
...	...

□

Example 6.5: We can even allow a constant as an expression in the SELECT clause. It might seem pointless to do so, but one application is to put some useful words into the output that SQL displays. The following query:

Case Insensitivity

SQL is *case insensitive*, meaning that it treats upper- and lower-case letters as the same letter. For example, although we have chosen to write keywords like `FROM` in capitals, it is equally proper to write this keyword as `From` or `from`, or even `FrOm`. Names of attributes, relations, aliases, and so on are similarly case insensitive. Only inside quotes does SQL make a distinction between upper- and lower-case letters. Thus, `'FROM'` and `'from'` are different character strings. Of course, neither is the keyword `FROM`.

```
SELECT title, length*0.016667 AS length, 'hrs.' AS inHours
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

produces tuples such as

title	*length*	*inHours*
Pretty Woman	1.98334	hrs.
...	...	...

We have arranged that the third column is called `inHours`, which fits with the column header `length` in the second column. Every tuple in the answer will have the constant `hrs.` in the third column, which gives the illusion of being the units attached to the value in the second column. □

6.1.2 Selection in SQL

The selection operator of relational algebra, and much more, is available through the `WHERE` clause of SQL. The expressions that may follow `WHERE` include conditional expressions like those found in common languages such as C or Java.

We may build expressions by comparing values using the six common comparison operators: =, <>, <, >, <=, and >=. The last four operators are as in C, but <> is the SQL symbol for "not equal to" (!= in C), and = in SQL is equality (== in C).

The values that may be compared include constants and attributes of the relations mentioned after `FROM`. We may also apply the usual arithmetic operators, $+$, $*$, and so on, to numeric values before we compare them. For instance, $(\texttt{year} - 1930) * (\texttt{year} - 1930) < 100$ is true for those years within 9 of 1930. We may apply the concatenation operator `||` to strings; for example `'foo' || 'bar'` has value `'foobar'`.

An example comparison is

```
studioName = 'Disney'
```

SQL Queries and Relational Algebra

The simple SQL queries that we have seen so far all have the form:

```
SELECT L
FROM R
WHERE C
```

in which L is a list of expressions, R is a relation, and C is a condition. The meaning of any such expression is the same as that of the relational-algebra expression

$$\pi_L(\sigma_C(R))$$

That is, we start with the relation in the FROM clause, apply to each tuple whatever condition is indicated in the WHERE clause, and then project onto the list of attributes and/or expressions in the SELECT clause.

in Example 6.1. The attribute studioName of the relation Movies is tested for equality against the constant 'Disney'. This constant is string-valued; strings in SQL are denoted by surrounding them with single quotes. Numeric constants, integers and reals, are also allowed, and SQL uses the common notations for reals such as -12.34 or 1.23E45.

The result of a comparison is a boolean value: either TRUE or FALSE.[2] Boolean values may be combined by the logical operators AND, OR, and NOT, with their expected meanings. For instance, we saw in Example 6.1 how two conditions could be combined by AND. The WHERE clause of this example evaluates to true if and only if both comparisons are satisfied; that is, the studio name is 'Disney' and the year is 1990. Here is an example of a query with a complex WHERE clause.

Example 6.6: Consider the query

```
SELECT title
FROM Movies
WHERE (year > 1970 OR length < 90) AND studioName = 'MGM';
```

This query asks for the titles of movies made by MGM Studios that either were made after 1970 or were less than 90 minutes long. Notice that comparisons can be grouped using parentheses. The parentheses are needed here because the precedence of logical operators in SQL is the same as in most other languages: AND takes precedence over OR, and NOT takes precedence over both. □

[2]Well there's a bit more to boolean values; see Section 6.1.7.

Representing Bit Strings

A string of bits is represented by `B` followed by a quoted string of 0's and 1's. Thus, `B'011'` represents the string of three bits, the first of which is 0 and the other two of which are 1. Hexadecimal notation may also be used, where an `X` is followed by a quoted string of hexadecimal digits (0 through 9, and *a* through *f*, with the latter representing "digits" 10 through 15). For instance, `X'7ff'` represents a string of twelve bits, a 0 followed by eleven 1's. Note that each hexadecimal digit represents four bits, and leading 0's are not suppressed.

6.1.3 Comparison of Strings

Two strings are equal if they are the same sequence of characters. Recall from Section 2.3.2 that strings can be stored as fixed-length strings, using `CHAR`, or variable-length strings, using `VARCHAR`. When comparing strings with different declarations, only the actual strings are compared; SQL ignores any "pad" characters that must be present in the database in order to give a string its required length.

When we compare strings by one of the "less than" operators, such as `<` or `>=`, we are asking whether one precedes the other in lexicographic order (i.e., in dictionary order, or alphabetically). That is, if $a_1a_2\cdots a_n$ and $b_1b_2\cdots b_m$ are two strings, then the first is "less than" the second if either $a_1 < b_1$, or if $a_1 = b_1$ and $a_2 < b_2$, or if $a_1 = b_1$, $a_2 = b_2$, and $a_3 < b_3$, and so on. We also say $a_1a_2\cdots a_n < b_1b_2\cdots b_m$ if $n < m$ and $a_1a_2\cdots a_n = b_1b_2\cdots b_n$; that is, the first string is a proper prefix of the second. For instance, `'fodder'` < `'foo'`, because the first two characters of each string are the same, `fo`, and the third character of `fodder` precedes the third character of `foo`. Also, `'bar'` < `'bargain'` because the former is a proper prefix of the latter.

6.1.4 Pattern Matching in SQL

SQL also provides the capability to compare strings on the basis of a simple pattern match. An alternative form of comparison expression is

```
    s LIKE p
```

where *s* is a string and *p* is a *pattern*, that is, a string with the optional use of the two special characters `%` and `_`. Ordinary characters in *p* match only themselves in *s*. But `%` in *p* can match any sequence of 0 or more characters in *s*, and `_` in *p* matches any one character in *s*. The value of this expression is true if and only if string *s* matches pattern *p*. Similarly, *s* `NOT LIKE` *p* is true if and only if string *s* does not match pattern *p*.

Example 6.7: We remember a movie "Star something," and we remember that the something has four letters. What could this movie be? We can retrieve all such names with the query:

```
SELECT title
FROM Movies
WHERE title LIKE 'Star ____';
```

This query asks if the title attribute of a movie has a value that is nine characters long, the first five characters being `Star` and a blank. The last four characters may be anything, since any sequence of four characters matches the four `_` symbols. The result of the query is the set of complete matching titles, such as *Star Wars* and *Star Trek*. □

Example 6.8: Let us search for all movies with a possessive ('s) in their titles. The desired query is

```
SELECT title
FROM Movies
WHERE title LIKE '%''s%';
```

To understand this pattern, we must first observe that the apostrophe, being the character that surrounds strings in SQL, cannot also represent itself. The convention taken by SQL is that two consecutive apostrophes in a string represent a single apostrophe and do not end the string. Thus, `''s` in a pattern is matched by a single apostrophe followed by an `s`.

The two `%` characters on either side of the `'s` match any strings whatsoever. Thus, any title with `'s` as a substring will match the pattern, and the answer to this query will include films such as *Logan's Run* or *Alice's Restaurant*. □

6.1.5 Dates and Times

Implementations of SQL generally support dates and times as special data types. These values are often representable in a variety of formats such as `05/14/1948` or `14 May 1948`. Here we shall describe only the SQL standard notation, which is very specific about format.

A *date* constant is represented by the keyword `DATE` followed by a quoted string of a special form. For example, `DATE '1948-05-14'` follows the required form. The first four characters are digits representing the year. Then come a hyphen and two digits representing the month. Note that, as in our example, a one-digit month is padded with a leading 0. Finally there is another hyphen and two digits representing the day. As with months, we pad the day with a leading 0 if that is necessary to make a two-digit number.

A *time* constant is represented similarly by the keyword `TIME` and a quoted string. This string has two digits for the hour, on the military (24-hour)

Escape Characters in LIKE expressions

What if the pattern we wish to use in a LIKE expression involves the characters % or _? Instead of having a particular character used as the escape character (e.g., the backslash in most UNIX commands), SQL allows us to specify any one character we like as the escape character for a single pattern. We do so by following the pattern by the keyword ESCAPE and the chosen escape character, in quotes. A character % or _ preceded by the escape character in the pattern is interpreted literally as that character, not as a symbol for any sequence of characters or any one character, respectively. For example,

```
s LIKE 'x%%x%' ESCAPE 'x'
```

makes x the escape character in the pattern x%%x%. The sequence x% is taken to be a single %. This pattern matches any string that begins and ends with the character %. Note that only the middle % has its "any string" interpretation.

clock. Then come a colon, two digits for the minute, another colon, and two digits for the second. If fractions of a second are desired, we may continue with a decimal point and as many significant digits as we like. For instance, TIME '15:00:02.5' represents the time at which all students will have left a class that ends at 3 PM: two and a half seconds past three o'clock.

Alternatively, time can be expressed as the number of hours and minutes ahead of (indicated by a plus sign) or behind (indicated by a minus sign) Greenwich Mean Time (GMT). For instance, TIME '12:00:00-8:00' represents noon in Pacific Standard Time, which is eight hours behind GMT.

To combine dates and times we use a value of type TIMESTAMP. These values consist of the keyword TIMESTAMP, a date value, a space, and a time value. Thus, TIMESTAMP '1948-05-14 12:00:00' represents noon on May 14, 1948.

We can compare dates or times using the same comparison operators we use for numbers or strings. That is, < on dates means that the first date is earlier than the second; < on times means that the first is earlier (within the same day) than the second.

6.1.6 Null Values and Comparisons Involving NULL

SQL allows attributes to have a special value NULL, which is called the *null value*. There are many different interpretations that can be put on null values. Here are some of the most common:

1. *Value unknown*: that is, "I know there is some value that belongs here but I don't know what it is." An unknown birthdate is an example.

2. *Value inapplicable*: "There is no value that makes sense here." For example, if we had a `spouse` attribute for the `MovieStar` relation, then an unmarried star might have `NULL` for that attribute, not because we don't know the spouse's name, but because there is none.

3. *Value withheld*: "We are not entitled to know the value that belongs here." For instance, an unlisted phone number might appear as `NULL` in the component for a `phone` attribute.

We saw in Section 5.2.7 how the use of the outerjoin operator of relational algebra produces null values in some components of tuples; SQL allows outerjoins and also produces `NULL`'s when a query involves outerjoins; see Section 6.3.8. There are other ways SQL produces `NULL`'s as well. For example, certain insertions of tuples create null values, as we shall see in Section 6.5.1.

In `WHERE` clauses, we must be prepared for the possibility that a component of some tuple we are examining will be `NULL`. There are two important rules to remember when we operate upon a `NULL` value.

1. When we operate on a `NULL` and any value, including another `NULL`, using an arithmetic operator like $\times$ or $+$, the result is `NULL`.

2. When we compare a `NULL` value and any value, including another `NULL`, using a comparison operator like `=` or `>`, the result is `UNKNOWN`. The value `UNKNOWN` is another truth-value, like `TRUE` and `FALSE`; we shall discuss how to manipulate truth-value `UNKNOWN` shortly.

However, we must remember that, although `NULL` is a value that can appear in tuples, it is *not* a constant. Thus, while the above rules apply when we try to operate on an expression whose value is `NULL`, we cannot use `NULL` explicitly as an operand.

Example 6.9: Let x have the value `NULL`. Then the value of $x+3$ is also `NULL`. However, `NULL` + 3 is not a legal SQL expression. Similarly, the value of $x = 3$ is `UNKNOWN`, because we cannot tell if the value of x, which is `NULL`, equals the value 3. However, the comparison `NULL` = 3 is not correct SQL. □

The correct way to ask if x has the value `NULL` is with the expression `x IS NULL`. This expression has the value `TRUE` if x has the value `NULL` and it has value `FALSE` otherwise. Similarly, `x IS NOT NULL` has the value `TRUE` unless the value of x is `NULL`.

6.1.7 The Truth-Value `UNKNOWN`

In Section 6.1.2 we assumed that the result of a comparison was either `TRUE` or `FALSE`, and these truth-values were combined in the obvious way using the logical operators `AND`, `OR`, and `NOT`. We have just seen that when `NULL` values

Pitfalls Regarding Nulls

It is tempting to assume that NULL in SQL can always be taken to mean "a value that we don't know but that surely exists." However, there are several ways that intuition is violated. For instance, suppose x is a component of some tuple, and the domain for that component is the integers. We might reason that $0 * x$ surely has the value 0, since no matter what integer x is, its product with 0 is 0. However, if x has the value NULL, rule (1) of Section 6.1.6 applies; the product of 0 and NULL is NULL. Similarly, we might reason that $x - x$ has the value 0, since whatever integer x is, its difference with itself is 0. However, again the rule about operations on nulls applies, and the result is NULL.

occur, comparisons can yield a third truth-value: UNKNOWN. We must now learn how the logical operators behave on combinations of all three truth-values.

The rule is easy to remember if we think of TRUE as 1 (i.e., fully true), FALSE as 0 (i.e., not at all true), and UNKNOWN as 1/2 (i.e., somewhere between true and false). Then:

1. The AND of two truth-values is the minimum of those values. That is, x AND y is FALSE if either x or y is FALSE; it is UNKNOWN if neither is FALSE but at least one is UNKNOWN, and it is TRUE only when both x and y are TRUE.

2. The OR of two truth-values is the maximum of those values. That is, x OR y is TRUE if either x or y is TRUE; it is UNKNOWN if neither is TRUE but at least one is UNKNOWN, and it is FALSE only when both are FALSE.

3. The negation of truth-value v is $1 - v$. That is, NOT x has the value TRUE when x is FALSE, the value FALSE when x is TRUE, and the value UNKNOWN when x has value UNKNOWN.

In Fig. 6.2 is a summary of the result of applying the three logical operators to the nine different combinations of truth-values for operands x and y. The value of the last operator, NOT, depends only on x.

SQL conditions, as appear in WHERE clauses of select-from-where statements, apply to each tuple in some relation, and for each tuple, one of the three truth values, TRUE, FALSE, or UNKNOWN is produced. However, only the tuples for which the condition has the value TRUE become part of the answer; tuples with either UNKNOWN or FALSE as value are excluded from the answer. That situation leads to another surprising behavior similar to that discussed in the box on "Pitfalls Regarding Nulls," as the next example illustrates.

Example 6.10: Suppose we ask about our running-example relation

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Figure 6.2: Truth table for three-valued logic

```
Movies(title, year, length, genre, studioName, producerC#)
```

the following query:

```
SELECT *
FROM Movies
WHERE length <= 120 OR length > 120;
```

Intuitively, we would expect to get a copy of the Movies relation, since each movie has a length that is either 120 or less or that is greater than 120.

However, suppose there are Movies tuples with NULL in the length component. Then both comparisons length <= 120 and length > 120 evaluate to UNKNOWN. The OR of two UNKNOWN's is UNKNOWN, by Fig. 6.2. Thus, for any tuple with a NULL in the length component, the WHERE clause evaluates to UNKNOWN. Such a tuple is *not* returned as part of the answer to the query. As a result, the true meaning of the query is "find all the Movies tuples with non-NULL lengths." □

6.1.8 Ordering the Output

We may ask that the tuples produced by a query be presented in sorted order. The order may be based on the value of any attribute, with ties broken by the value of a second attribute, remaining ties broken by a third, and so on, as in the τ operation of Section 5.2.6. To get output in sorted order, we may add to the select-from-where statement a clause:

```
ORDER BY <list of attributes>
```

The order is by default ascending, but we can get the output highest-first by appending the keyword DESC (for "descending") to an attribute. Similarly, we can specify ascending order with the keyword ASC, but that word is unnecessary.

The ORDER BY clause follows the WHERE clause and any other clauses (i.e., the optional GROUP BY and HAVING clauses, which are introduced in Section 6.4).

The ordering is performed on the result of the FROM, WHERE, and other clauses, just before we apply the SELECT clause. The tuples of this result are then sorted by the attributes in the list of the ORDER BY clause, and then passed to the SELECT clause for processing in the normal manner.

Example 6.11: The following is a rewrite of our original query of Example 6.1, asking for the Disney movies of 1990 from the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

To get the movies listed by length, shortest first, and among movies of equal length, alphabetically, we can say:

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length, title;
```

A subtlety of ordering is that all the attributes of Movies are available at the time of sorting, even if they are not part of the SELECT clause. Thus, we could replace SELECT * by SELECT producerC#, and the query would still be legal. □

An additional option in ordering is that the list following ORDER BY can include expressions, just as the SELECT clause can. For instance, we can order the tuples of a relation $R(A, B)$ by the sum of the two components of the tuples, highest first, with:

```
SELECT *
FROM R
ORDER BY A+B DESC;
```

6.1.9 Exercises for Section 6.1

Exercise 6.1.1: If a query has a SELECT clause

```
SELECT A B
```

how do we know whether A and B are two different attributes or B is an alias of A?

Exercise 6.1.2: Write the following queries, based on our running movie database example

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

in SQL.

a) Find the address of MGM studios.

b) Find Sandra Bullock's birthdate.

c) Find all the stars that appeared either in a movie made in 1980 or a movie with "Love" in the title.

d) Find all executives worth at least $10,000,000.

e) Find all the stars who either are male or live in Malibu (have string `Malibu` as a part of their address).

Exercise 6.1.3: Write the following queries in SQL. They refer to the database schema of Exercise 2.4.1:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

Show the result of your queries using the data from Exercise 2.4.1.

a) Find the model number, speed, and hard-disk size for all PC's whose price is under $1000.

b) Do the same as (a), but rename the `speed` column `gigahertz` and the `hd` column `gigabytes`.

c) Find the manufacturers of printers.

d) Find the model number, memory size, and screen size for laptops costing more than $1500.

e) Find all the tuples in the `Printer` relation for color printers. Remember that `color` is a boolean-valued attribute.

f) Find the model number and hard-disk size for those PC's that have a speed of 3.2 and a price less than $2000.

Exercise 6.1.4: Write the following queries based on the database schema of Exercise 2.4.3:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

and show the result of your query on the data of Exercise 2.4.3.

a) Find the class name and country for all classes with at least 10 guns.

b) Find the names of all ships launched prior to 1918, but call the resulting column `shipName`.

c) Find the names of ships sunk in battle and the name of the battle in which they were sunk.

d) Find all ships that have the same name as their class.

e) Find the names of all ships that begin with the letter "R."

! f) Find the names of all ships whose name consists of three or more words (e.g., King George V).

Exercise 6.1.5: Let a and b be integer-valued attributes that may be `NULL` in some tuples. For each of the following conditions (as may appear in a `WHERE` clause), describe exactly the set of (a, b) tuples that satisfy the condition, including the case where a and/or b is `NULL`.

a) `a = 10 OR b = 20`

b) `a = 10 AND b = 20`

c) `a < 10 OR a >= 10`

! d) `a = b`

! e) `a <= b`

! **Exercise 6.1.6:** In Example 6.10 we discussed the query

```
SELECT *
FROM Movies
WHERE length <= 120 OR length > 120;
```

which behaves unintuitively when the length of a movie is `NULL`. Find a simpler, equivalent query, one with a single condition in the `WHERE` clause (no `AND` or `OR` of conditions).

6.2 Queries Involving More Than One Relation

Much of the power of relational algebra comes from its ability to combine two or more relations through joins, products, unions, intersections, and differences. We get all of these operations in SQL. The set-theoretic operations — union, intersection, and difference — appear directly in SQL, as we shall learn in Section 6.2.5. First, we shall learn how the select-from-where statement of SQL allows us to perform products and joins.

6.2.1 Products and Joins in SQL

SQL has a simple way to couple relations in one query: list each relation in the `FROM` clause. Then, the `SELECT` and `WHERE` clauses can refer to the attributes of any of the relations in the `FROM` clause.

Example 6.12: Suppose we want to know the name of the producer of *Star Wars.* To answer this question we need the following two relations from our running example:

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

The producer certificate number is given in the `Movies` relation, so we can do a simple query on `Movies` to get this number. We could then do a second query on the relation `MovieExec` to find the name of the person with that certificate number.

However, we can phrase both these steps as one query about the pair of relations `Movies` and `MovieExec` as follows:

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

This query asks us to consider all pairs of tuples, one from `Movies` and the other from `MovieExec`. The conditions on this pair are stated in the `WHERE` clause:

1. The `title` component of the tuple from `Movies` must have value `'Star Wars'`.

2. The `producerC#` attribute of the `Movies` tuple must be the same certificate number as the `cert#` attribute in the `MovieExec` tuple. That is, these two tuples must refer to the same producer.

Whenever we find a pair of tuples satisfying both conditions, we produce the `name` attribute of the tuple from `MovieExec` as part of the answer. If the data is what we expect, the only time both conditions will be met is when the tuple from `Movies` is for *Star Wars*, and the tuple from `MovieExec` is for George Lucas. Then and only then will the title be correct and the certificate numbers agree. Thus, `George Lucas` should be the only value produced. This process is suggested in Fig. 6.3. We take up in more detail how to interpret multirelation queries in Section 6.2.4. □

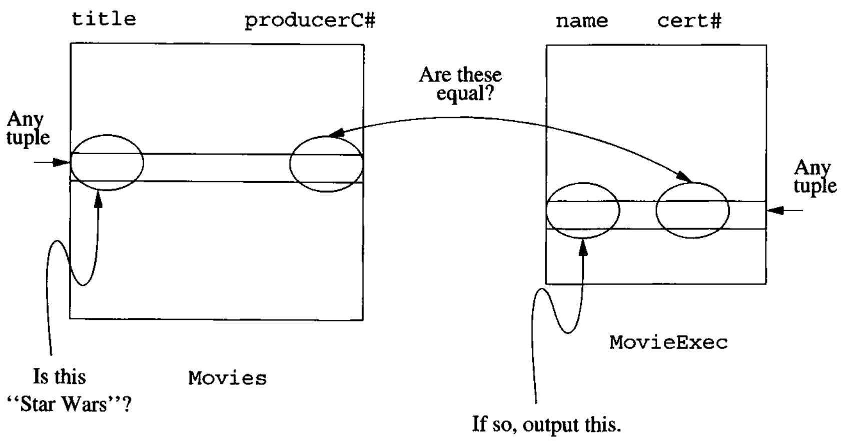

Figure 6.3: The query of Example 6.12 asks us to pair every tuple of `Movies` with every tuple of `MovieExec` and test two conditions

6.2.2 Disambiguating Attributes

Sometimes we ask a query involving several relations, and among these relations are two or more attributes with the same name. If so, we need a way to indicate which of these attributes is meant by a use of their shared name. SQL solves this problem by allowing us to place a relation name and a dot in front of an attribute. Thus $R.A$ refers to the attribute A of relation R.

Example 6.13: The two relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

each have attributes `name` and `address`. Suppose we wish to find pairs consisting of a star and an executive with the same address. The following query does the job.

```
SELECT MovieStar.name, MovieExec.name
FROM MovieStar, MovieExec
WHERE MovieStar.address = MovieExec.address;
```

In this query, we look for a pair of tuples, one from `MovieStar` and the other from `MovieExec`, such that their address components agree. The `WHERE` clause enforces the requirement that the `address` attributes from each of the two tuples agree. Then, for each matching pair of tuples, we extract the two `name` attributes, first from the `MovieStar` tuple and then from the other. The result would be a set of pairs such as

MovieStar.name	*MovieExec.name*
Jane Fonda	Ted Turner
...	...

□

The relation name, followed by a dot, is permissible even in situations where there is no ambiguity. For instance, we are free to write the query of Example 6.12 as

```
SELECT MovieExec.name
FROM Movies, MovieExec
WHERE Movie.title = 'Star Wars'
      AND Movie.producerC# = MovieExec.cert#;
```

Alternatively, we may use relation names and dots in front of any subset of the attributes in this query.

6.2.3 Tuple Variables

Disambiguating attributes by prefixing the relation name works as long as the query involves combining several different relations. However, sometimes we need to ask a query that involves two or more tuples from the same relation. We may list a relation R as many times as we need to in the FROM clause, but we need a way to refer to each occurrence of R. SQL allows us to define, for each occurrence of R in the FROM clause, an "alias" which we shall refer to as a *tuple variable*. Each use of R in the FROM clause is followed by the (optional) keyword AS and the name of the tuple variable; we shall generally omit the AS in this context.

In the SELECT and WHERE clauses, we can disambiguate attributes of R by preceding them by the appropriate tuple variable and a dot. Thus, the tuple variable serves as another name for relation R and can be used in its place when we wish.

Example 6.14: While Example 6.13 asked for a star and an executive sharing an address, we might similarly want to know about two stars who share an address. The query is essentially the same, but now we must think of two tuples chosen from relation MovieStar, rather than tuples from each of MovieStar and MovieExec. Using tuple variables as aliases for two uses of MovieStar, we can write the query as

```
SELECT Star1.name, Star2.name
FROM MovieStar Star1, MovieStar Star2
WHERE Star1.address = Star2.address
      AND Star1.name < Star2.name;
```

Tuple Variables and Relation Names

Technically, references to attributes in SELECT and WHERE clauses are *always* to a tuple variable. However, if a relation appears only once in the FROM clause, then we can use the relation name as its own tuple variable. Thus, we can see a relation name R in the FROM clause as shorthand for R AS R. Furthermore, as we have seen, when an attribute belongs unambiguously to one relation, the relation name (tuple variable) may be omitted.

We see in the FROM clause the declaration of two tuple variables, `Star1` and `Star2`; each is an alias for relation `MovieStar`. The tuple variables are used in the SELECT clause to refer to the `name` components of the two tuples. These aliases are also used in the WHERE clause to say that the two `MovieStar` tuples represented by `Star1` and `Star2` have the same value in their `address` components.

The second condition in the WHERE clause, `Star1.name < Star2.name`, says that the name of the first star precedes the name of the second star alphabetically. If this condition were omitted, then tuple variables `Star1` and `Star2` could both refer to the same tuple. We would find that the two tuple variables referred to tuples whose `address` components are equal, of course, and thus produce each star name paired with itself.[3] The second condition also forces us to produce each pair of stars with a common address only once, in alphabetical order. If we used <> (not-equal) as the comparison operator, then we would produce pairs of married stars twice, like

Star1.name	*Star2.name*
Paul Newman	Joanne Woodward
Joanne Woodward	Paul Newman
...	...

□

6.2.4 Interpreting Multirelation Queries

There are several ways to define the meaning of the select-from-where expressions that we have just covered. All are *equivalent*, in the sense that they each give the same answer for each query applied to the same relation instances. We shall consider each in turn.

[3] A similar problem occurs in Example 6.13 when the same individual is both a star and an executive. We could solve that problem by requiring that the two names be unequal.

Nested Loops

The semantics that we have implicitly used in examples so far is that of tuple variables. Recall that a tuple variable ranges over all tuples of the corresponding relation. A relation name that is not aliased is also a tuple variable ranging over the relation itself, as we mentioned in the box on "Tuple Variables and Relation Names." If there are several tuple variables, we may imagine nested loops, one for each tuple variable, in which the variables each range over the tuples of their respective relations. For each assignment of tuples to the tuple variables, we decide whether the WHERE clause is true. If so, we produce a tuple consisting of the values of the expressions following SELECT; note that each term is given a value by the current assignment of tuples to tuple variables. This query-answering algorithm is suggested by Fig. 6.4.

```
LET the tuple variables in the from-clause range over
        relations $R_1, R_2, \ldots, R_n$;
FOR each tuple $t_1$ in relation $R_1$ DO
    FOR each tuple $t_2$ in relation $R_2$ DO
        ...
            FOR each tuple $t_n$ in relation $R_n$ DO
                IF the where-clause is satisfied when the values
                from $t_1, t_2, \ldots, t_n$ are substituted for all
                attribute references THEN
                    evaluate the expressions of the select-clause
                    according to $t_1, t_2, \ldots, t_n$ and produce the
                    tuple of values that results.
```

Figure 6.4: Answering a simple SQL query

Parallel Assignment

There is an equivalent definition in which we do not explicitly create nested loops ranging over the tuple variables. Rather, we consider in arbitrary order, or in parallel, all possible assignments of tuples from the appropriate relations to the tuple variables. For each such assignment, we consider whether the WHERE clause becomes true. Each assignment that produces a true WHERE clause contributes a tuple to the answer; that tuple is constructed from the attributes of the SELECT clause, evaluated according to that assignment.

Conversion to Relational Algebra

A third approach is to relate the SQL query to relational algebra. We start with the tuple variables in the FROM clause and take the Cartesian product of their relations. If two tuple variables refer to the same relation, then this relation appears twice in the product, and we rename its attributes so all attributes have

An Unintuitive Consequence of SQL Semantics

Suppose R, S, and T are unary (one-component) relations, each having attribute A alone, and we wish to find those elements that are in R and also in either S or T (or both). That is, we want to compute $R \cap (S \cup T)$. We might expect the following SQL query would do the job.

```
SELECT R.A
FROM R, S, T
WHERE R.A = S.A OR R.A = T.A;
```

However, consider the situation in which T is empty. Since then $R.A = T.A$ can never be satisfied, we might expect the query to produce exactly $R \cap S$, based on our intuition about how "OR" operates. Yet whichever of the three equivalent definitions of Section 6.2.4 one prefers, we find that the result is empty, regardless of how many elements R and S have in common. If we use the nested-loop semantics of Figure 6.4, then we see that the loop for tuple variable T iterates 0 times, since there are no tuples in the relation for the tuple variable to range over. Thus, the if-statement inside the for-loops never executes, and nothing can be produced. Similarly, if we look for assignments of tuples to the tuple variables, there is no way to assign a tuple to T, so no assignments exist. Finally, if we use the Cartesian-product approach, we start with $R \times S \times T$, which is empty because T is empty.

unique names. Similarly, attributes of the same name from different relations are renamed to avoid ambiguity.

Having created the product, we apply a selection operator to it by converting the WHERE clause to a selection condition in the obvious way. That is, each attribute reference in the WHERE clause is replaced by the attribute of the product to which it corresponds. Finally, we create from the SELECT clause a list of expressions for a final (extended) projection operation. As we did for the WHERE clause, we interpret each attribute reference in the SELECT clause as the corresponding attribute in the product of relations.

Example 6.15: Let us convert the query of Example 6.14 to relational algebra. First, there are two tuple variables in the FROM clause, both referring to relation MovieStar. Thus, our expression (without the necessary renaming) begins:

$$\texttt{MovieStar} \times \texttt{MovieStar}$$

The resulting relation has eight attributes, the first four correspond to attributes name, address, gender, and birthdate from the first copy of relation MovieStar, and the second four correspond to the same attributes from the other copy of `MovieStar`. We could create names for these attributes with a dot and the aliasing tuple variable — e.g., `Star1.gender` — but for succinctness, let us invent new symbols and call the attributes simply $A_1, A_2, \ldots, A_8$. Thus, A_1 corresponds to `Star1.name`, A_5 corresponds to `Star2.name`, and so on.

Under this naming strategy for attributes, the selection condition obtained from the `WHERE` clause is $A_2 = A_6$ and $A_1 < A_5$. The projection list is A_1, A_5. Thus,

$$\pi_{A_1,A_5}\Big(\sigma_{A_2=A_6 \texttt{ AND } A_1<A_5}\big(\rho_{M(A_1,A_2,A_3,A_4)}(\texttt{MovieStar}) \times \rho_{N(A_5,A_6,A_7,A_8)}(\texttt{MovieStar})\big)\Big)$$

renders the entire query in relational algebra. □

6.2.5 Union, Intersection, and Difference of Queries

Sometimes we wish to combine relations using the set operations of relational algebra: union, intersection, and difference. SQL provides corresponding operators that apply to the results of queries, provided those queries produce relations with the same list of attributes and attribute types. The keywords used are `UNION`, `INTERSECT`, and `EXCEPT` for $\cup$, $\cap$, and $-$, respectively. Words like `UNION` are used between two queries, and those queries must be parenthesized.

Example 6.16: Suppose we wanted the names and addresses of all female movie stars who are also movie executives with a net worth over $10,000,000. Using the following two relations:

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

we can write the query as in Fig. 6.5. Lines (1) through (3) produce a relation whose schema is `(name, address)` and whose tuples are the names and addresses of all female movie stars.

```
1)  (SELECT name, address
2)   FROM MovieStar
3)   WHERE gender = 'F')
4)       INTERSECT
5)  (SELECT name, address
6)   FROM MovieExec
7)   WHERE netWorth > 10000000);
```

Figure 6.5: Intersecting female movie stars with rich executives

Similarly, lines (5) through (7) produce the set of "rich" executives, those with net worth over $10,000,000. This query also yields a relation whose schema

Readable SQL Queries

Generally, one writes SQL queries so that each important keyword like FROM or WHERE starts a new line. This style offers the reader visual clues to the structure of the query. However, when a query or subquery is short, we shall sometimes write it out on a single line, as we did in Example 6.17. That style, keeping a complete query compact, also offers good readability.

has the attributes name and address only. Since the two schemas are the same, we can intersect them, and we do so with the operator of line (4). □

Example 6.17: In a similar vein, we could take the difference of two sets of persons, each selected from a relation. The query

```
(SELECT name, address FROM MovieStar)
    EXCEPT
(SELECT name, address FROM MovieExec);
```

gives the names and addresses of movie stars who are not also movie executives, regardless of gender or net worth. □

In the two examples above, the attributes of the relations whose intersection or difference we took were conveniently the same. However, if necessary to get a common set of attributes, we can rename attributes as in Example 6.3.

Example 6.18: Suppose we wanted all the titles and years of movies that appeared in either the Movies or StarsIn relation of our running example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
```

Ideally, these sets of movies would be the same, but in practice it is common for relations to diverge; for instance we might have movies with no listed stars or a StarsIn tuple that mentions a movie not found in the Movies relation.[4] Thus, we might write

```
(SELECT title, year FROM Movie)
    UNION
(SELECT movieTitle AS title, movieYear AS year FROM StarsIn);
```

The result would be all movies mentioned in either relation, with title and year as the attributes of the resulting relation. □

[4]There are ways to prevent this divergence; see Section 7.1.1.

6.2.6 Exercises for Section 6.2

Exercise 6.2.1: Using the database schema of our running movie example

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

write the following queries in SQL.

a) Who were the male stars in *Titanic*?

b) Which stars appeared in movies produced by MGM in 1995?

c) Who is the president of MGM studios?

! d) Which movies are longer than *Gone With the Wind*?

! e) Which executives are worth more than Merv Griffin?

Exercise 6.2.2: Write the following queries, based on the database schema

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

of Exercise 2.4.1, and evaluate your queries using the data of that exercise.

a) Give the manufacturer and speed of laptops with a hard disk of at least thirty gigabytes.

b) Find the model number and price of all products (of any type) made by manufacturer B.

c) Find those manufacturers that sell Laptops, but not PC's.

! d) Find those hard-disk sizes that occur in two or more PC's.

! e) Find those pairs of PC models that have both the same speed and RAM. A pair should be listed only once; e.g., list (i, j) but not (j, i).

!! f) Find those manufacturers of at least two different computers (PC's or laptops) with speeds of at least 3.0.

Exercise 6.2.3: Write the following queries, based on the database schema

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

of Exercise 2.4.3, and evaluate your queries using the data of that exercise.

a) Find the ships heavier than 35,000 tons.

b) List the name, displacement, and number of guns of the ships engaged in the battle of Guadalcanal.

c) List all the ships mentioned in the database. (Remember that all these ships may not appear in the `Ships` relation.)

! d) Find those countries that have both battleships and battlecruisers.

! e) Find those ships that were damaged in one battle, but later fought in another.

! f) Find those battles with at least three ships of the same country.

! **Exercise 6.2.4:** A general form of relational-algebra query is

$$\pi_L\Big(\sigma_C(R_1 \times R_2 \times \cdots \times R_n)\Big)$$

Here, L is an arbitrary list of attributes, and C is an arbitrary condition. The list of relations $R_1, R_2, \ldots, R_n$ may include the same relation repeated several times, in which case appropriate renaming may be assumed applied to the R_i's. Show how to express any query of this form in SQL.

! **Exercise 6.2.5:** Another general form of relational-algebra query is

$$\pi_L\Big(\sigma_C(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n)\Big)$$

The same assumptions as in Exercise 6.2.4 apply here; the only difference is that the natural join is used instead of the product. Show how to express any query of this form in SQL.

6.3 Subqueries

In SQL, one query can be used in various ways to help in the evaluation of another. A query that is part of another is called a *subquery*. Subqueries can have subqueries, and so on, down as many levels as we wish. We already saw one example of the use of subqueries; in Section 6.2.5 we built a union, intersection, or difference query by connecting two subqueries to form the whole query. There are a number of other ways that subqueries can be used:

1. Subqueries can return a single constant, and this constant can be compared with another value in a WHERE clause.

2. Subqueries can return relations that can be used in various ways in WHERE clauses.

3. Subqueries can appear in FROM clauses, followed by a tuple variable that represents the tuples in the result of the subquery.

6.3.1 Subqueries that Produce Scalar Values

An atomic value that can appear as one component of a tuple is referred to as a *scalar.* A select-from-where expression can produce a relation with any number of attributes in its schema, and there can be any number of tuples in the relation. However, often we are only interested in values of a single attribute. Furthermore, sometimes we can deduce from information about keys, or from other information, that there will be only a single value produced for that attribute.

If so, we can use this select-from-where expression, surrounded by parentheses, as if it were a constant. In particular, it may appear in a WHERE clause any place we would expect to find a constant or an attribute representing a component of a tuple. For instance, we may compare the result of such a subquery to a constant or attribute.

Example 6.19: Let us recall Example 6.12, where we asked for the producer of *Star Wars.* We had to query the two relations

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

because only the former has movie title information and only the latter has producer names. The information is linked by "certificate numbers." These numbers uniquely identify producers. The query we developed is:

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

There is another way to look at this query. We need the Movies relation only to get the certificate number for the producer of *Star Wars.* Once we have it, we can query the relation MovieExec to find the name of the person with this certificate. The first problem, getting the certificate number, can be written as a subquery, and the result, which we expect will be a single value, can be used in the "main" query to achieve the same effect as the query above. This query is shown in Fig. 6.6.

Lines (4) through (6) of Fig. 6.6 are the subquery. Looking only at this simple query by itself, we see that the result will be a unary relation with

```
1)  SELECT name
2)  FROM MovieExec
3)  WHERE cert# =
4)      (SELECT producerC#
5)       FROM Movies
6)       WHERE title = 'Star Wars'
        );
```

Figure 6.6: Finding the producer of *Star Wars* by using a nested subquery

attribute producerC#, and we expect to find only one tuple in this relation. The tuple will look like (12345), that is, a single component with some integer, perhaps 12345 or whatever George Lucas' certificate number is. If zero tuples or more than one tuple is produced by the subquery of lines (4) through (6), it is a run-time error.

Having executed this subquery, we can then execute lines (1) through (3) of Fig. 6.6, as if the value 12345 replaced the entire subquery. That is, the "main" query is executed as if it were

```
SELECT name
FROM MovieExec
WHERE cert# = 12345;
```

The result of this query should be George Lucas. □

6.3.2 Conditions Involving Relations

There are a number of SQL operators that we can apply to a relation R and produce a boolean result. However, the relation R must be expressed as a subquery. As a trick, if we want to apply these operators to a stored table Foo, we can use the subquery (SELECT * FROM Foo). The same trick works for union, intersection, and difference of relations. Notice that those operators, introduced in Section 6.2.5 are applied to two subqueries.

Some of the operators below — IN, ALL, and ANY — will be explained first in their simple form where a scalar value s is involved. In this situation, the subquery R is required to produce a one-column relation. Here are the definitions of the operators:

1. EXISTS R is a condition that is true if and only if R is not empty.

2. s IN R is true if and only if s is equal to one of the values in R. Likewise, s NOT IN R is true if and only if s is equal to no value in R. Here, we assume R is a unary relation. We shall discuss extensions to the IN and NOT IN operators where R has more than one attribute in its schema and s is a tuple in Section 6.3.3.

3. s > ALL R is true if and only if s is greater than every value in unary relation R. Similarly, the > operator could be replaced by any of the other five comparison operators, with the analogous meaning: s stands in the stated relationship to every tuple in R. For instance, s <> ALL R is the same as s NOT IN R.

4. s > ANY R is true if and only if s is greater than at least one value in unary relation R. Similarly, any of the other five comparisons could be used in place of >, with the meaning that s stands in the stated relationship to at least one tuple of R. For instance, s = ANY R is the same as s IN R.

The EXISTS, ALL, and ANY operators can be negated by putting NOT in front of the entire expression, just like any other boolean-valued expression. Thus, NOT EXISTS R is true if and only if R is empty. NOT s >= ALL R is true if and only if s is not the maximum value in R, and NOT s > ANY R is true if and only if s is the minimum value in R. We shall see several examples of the use of these operators shortly.

6.3.3 Conditions Involving Tuples

A tuple in SQL is represented by a parenthesized list of scalar values. Examples are (123, 'foo') and (name, address, networth). The first of these has constants as components; the second has attributes as components. Mixing of constants and attributes is permitted.

If a tuple t has the same number of components as a relation R, then it makes sense to compare t and R in expressions of the type listed in Section 6.3.2. Examples are t IN R or t <> ANY R. The latter comparison means that there is some tuple in R other than t. Note that when comparing a tuple with members of a relation R, we must compare components using the assumed standard order for the attributes of R.

```
1)  SELECT name
2)  FROM MovieExec
3)  WHERE cert# IN
4)      (SELECT producerC#
5)       FROM Movies
6)       WHERE (title, year) IN
7)           (SELECT movieTitle, movieYear
8)            FROM StarsIn
9)            WHERE starName = 'Harrison Ford'
           )
      );
```

Figure 6.7: Finding the producers of Harrison Ford's movies

Example 6.20: In Fig. 6.7 is a SQL query on the three relations

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
```

asking for all the producers of movies in which Harrison Ford stars. It consists of a "main" query, a query nested within that, and a third query nested within the second.

We should analyze any query with subqueries from the inside out. Thus, let us start with the innermost nested subquery: lines (7) through (9). This query examines the tuples of the relation `StarsIn` and finds all those tuples whose `starName` component is `'Harrison Ford'`. The titles and years of those movies are returned by this subquery. Recall that title and year, not title alone, is the key for movies, so we need to produce tuples with both attributes to identify a movie uniquely. Thus, we would expect the value produced by lines (7) through (9) to look something like Fig. 6.8.

title	*year*
Star Wars	1977
Raiders of the Lost Ark	1981
The Fugitive	1993
...	...

Figure 6.8: Title-year pairs returned by inner subquery

Now, consider the middle subquery, lines (4) through (6). It searches the `Movies` relation for tuples whose title and year are in the relation suggested by Fig. 6.8. For each tuple found, the producer's certificate number is returned, so the result of the middle subquery is the set of certificates of the producers of Harrison Ford's movies.

Finally, consider the "main" query of lines (1) through (3). It examines the tuples of the `MovieExec` relation to find those whose `cert#` component is one of the certificates in the set returned by the middle subquery. For each of these tuples, the name of the producer is returned, giving us the set of producers of Harrison Ford's movies, as desired. □

Incidentally, the nested query of Fig. 6.7 can, like many nested queries, be written as a single select-from-where expression with relations in the `FROM` clause for each of the relations mentioned in the main query or a subquery. The `IN` relationships are replaced by equalities in the `WHERE` clause. For instance, the query of Fig. 6.9 is essentially that of Fig. 6.7. There is a difference regarding the way duplicate occurrences of a producer — e.g., George Lucas — are handled, as we shall discuss in Section 6.4.1.

```
SELECT name
FROM MovieExec, Movies, StarsIn
WHERE cert# = producerC# AND
      title = movieTitle AND
      year = movieYear AND
      starName = 'Harrison Ford';
```

Figure 6.9: Ford's producers without nested subqueries

6.3.4 Correlated Subqueries

The simplest subqueries can be evaluated once and for all, and the result used in a higher-level query. A more complicated use of nested subqueries requires the subquery to be evaluated many times, once for each assignment of a value to some term in the subquery that comes from a tuple variable outside the subquery. A subquery of this type is called a *correlated* subquery. Let us begin our study with an example.

Example 6.21: We shall find the titles that have been used for two or more movies. We start with an outer query that looks at all tuples in the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

For each such tuple, we ask in a subquery whether there is a movie with the same title and a greater year. The entire query is shown in Fig. 6.10.

As with other nested queries, let us begin at the innermost subquery, lines (4) through (6). If `Old.title` in line (6) were replaced by a constant string such as `'King Kong'`, we would understand it quite easily as a query asking for the year or years in which movies titled *King Kong* were made. The present subquery differs little. The only problem is that we don't know what value `Old.title` has. However, as we range over `Movies` tuples of the outer query of lines (1) through (3), each tuple provides a value of `Old.title`. We then execute the query of lines (4) through (6) with this value for `Old.title` to decide the truth of the `WHERE` clause that extends from lines (3) through (6).

```
1)  SELECT title
2)  FROM Movies Old
3)  WHERE year < ANY
4)      (SELECT year
5)       FROM Movies
6)       WHERE title = Old.title
        );
```

Figure 6.10: Finding movie titles that appear more than once

The condition of line (3) is true if any movie with the same title as `Old.title` has a later year than the movie in the tuple that is the current value of tuple variable `Old`. This condition is true unless the year in the tuple `Old` is the last year in which a movie of that title was made. Consequently, lines (1) through (3) produce a title one fewer times than there are movies with that title. A movie made twice will be listed once, a movie made three times will be listed twice, and so on.[5] □

When writing a correlated query it is important that we be aware of the *scoping rules* for names. In general, an attribute in a subquery belongs to one of the tuple variables in that subquery's `FROM` clause if some tuple variable's relation has that attribute in its schema. If not, we look at the immediately surrounding subquery, then to the one surrounding that, and so on. Thus, `year` on line (4) and `title` on line (6) of Fig. 6.10 refer to the attributes of the tuple variable that ranges over all the tuples of the copy of relation `Movies` introduced on line (5) — that is, the copy of the `Movies` relation addressed by the subquery of lines (4) through (6).

However, we can arrange for an attribute to belong to another tuple variable if we prefix it by that tuple variable and a dot. That is why we introduced the alias `Old` for the `Movies` relation of the outer query, and why we refer to `Old.title` in line (6). Note that if the two relations in the `FROM` clauses of lines (2) and (5) were different, we would not need an alias. Rather, in the subquery we could refer directly to attributes of a relation mentioned in line (2).

6.3.5 Subqueries in FROM Clauses

Another use for subqueries is as relations in a `FROM` clause. In a `FROM` list, instead of a stored relation, we may use a parenthesized subquery. Since we don't have a name for the result of this subquery, we must give it a tuple-variable alias. We then refer to tuples in the result of the subquery as we would tuples in any relation that appears in the `FROM` list.

Example 6.22: Let us reconsider the problem of Example 6.20, where we wrote a query that finds the producers of Harrison Ford's movies. Suppose we had a relation that gave the certificates of the producers of those movies. It would then be a simple matter to look up the names of those producers in the relation `MovieExec`. Figure 6.11 is such a query.

Lines (2) through (7) are the `FROM` clause of the outer query. In addition to the relation `MovieExec`, it has a subquery. That subquery joins `Movies` and `StarsIn` on lines (3) through (5), adds the condition that the star is Harrison Ford on line (6), and returns the set of producers of the movies at line (2). This set is given the alias `Prod` on line (7).

[5]This example is the first occasion on which we've been reminded that relations in SQL are bags, not sets. There are several ways that duplicates may crop up in SQL relations. We shall discuss the matter in detail in Section 6.4.

```
1)  SELECT name
2)  FROM MovieExec, (SELECT producerC#
3)                   FROM Movies, StarsIn
4)                   WHERE title = movieTitle AND
5)                         year = movieYear AND
6)                         starName = 'Harrison Ford'
7)                  ) Prod
8)  WHERE cert# = Prod.producerC#;
```

Figure 6.11: Finding the producers of Ford's movies using a subquery in the `FROM` clause

At line (8), the relations `MovieExec` and the subquery aliased `Prod` are joined with the requirement that the certificate numbers be the same. The names of the producers from `MovieExec` that have certificates in the set aliased by `Prod` is returned at line (1). □

6.3.6 SQL Join Expressions

We can construct relations by a number of variations on the join operator applied to two relations. These variants include products, natural joins, theta-joins, and outerjoins. The result can stand as a query by itself. Alternatively, all these expressions, since they produce relations, may be used as subqueries in the `FROM` clause of a select-from-where expression. These expressions are principally shorthands for more complex select-from-where queries (see Exercise 6.3.11).

The simplest form of join expression is a *cross join*; that term is a synonym for what we called a Cartesian product or just "product" in Section 2.4.7. For instance, if we want the product of the two relations

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
```

we can say

```
Movies CROSS JOIN StarsIn;
```

and the result will be a nine-column relation with all the attributes of `Movies` and `StarsIn`. Every pair consisting of one tuple of `Movies` and one tuple of `StarsIn` will be a tuple of the resulting relation.

The attributes in the product relation can be called $R.A$, where R is one of the two joined relations and A is one of its attributes. If only one of the relations has an attribute named A, then the R and dot can be dropped, as usual. In this instance, since `Movies` and `StarsIn` have no common attributes, the nine attribute names suffice in the product.

However, the product by itself is rarely a useful operation. A more conventional theta-join is obtained with the keyword `ON`. We put `JOIN` between two relation names R and S and follow them by ON and a condition. The meaning of JOIN...ON is that the product of $R \times S$ is followed by a selection for whatever condition follows ON.

Example 6.23: Suppose we want to join the relations

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
```

with the condition that the only tuples to be joined are those that refer to the same movie. That is, the titles and years from both relations must be the same. We can ask this query by

```
Movies JOIN StarsIn ON
        title = movieTitle AND year = movieYear;
```

The result is again a nine-column relation with the obvious attribute names. However, now a tuple from Movies and one from StarsIn combine to form a tuple of the result only if the two tuples agree on both the title and year. As a result, two of the columns are redundant, because every tuple of the result will have the same value in both the title and movieTitle components and will have the same value in both year and movieYear.

If we are concerned with the fact that the join above has two redundant components, we can use the whole expression as a subquery in a FROM clause and use a SELECT clause to remove the undesired attributes. Thus, we could write

```
SELECT title, year, length, genre, studioName,
        producerC#, starName
FROM Movies JOIN StarsIn ON
        title = movieTitle AND year = movieYear;
```

to get a seven-column relation which is the Movies relation's tuples, each extended in all possible ways with a star of that movie. □

6.3.7 Natural Joins

As we recall from Section 2.4.8, a natural join differs from a theta-join in that:

1. The join condition is that all pairs of attributes from the two relations having a common name are equated, and there are no other conditions.

2. One of each pair of equated attributes is projected out.

The SQL natural join behaves exactly this way. Keywords NATURAL JOIN appear between the relations to express the $\bowtie$ operator.

Example 6.24: Suppose we want to compute the natural join of the relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

The result will be a relation whose schema includes attributes `name` and `address` plus all the attributes that appear in one or the other of the two relations. A tuple of the result will represent an individual who is both a star and an executive and will have all the information pertinent to either: a name, address, gender, birthdate, certificate number, and net worth. The expression

```
MovieStar NATURAL JOIN MovieExec;
```

succinctly describes the desired relation. □

6.3.8 Outerjoins

The outerjoin operator was introduced in Section 5.2.7 as a way to augment the result of a join by the dangling tuples, padded with null values. In SQL, we can specify an outerjoin; `NULL` is used as the null value.

Example 6.25: Suppose we wish to take the outerjoin of the two relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

SQL refers to the standard outerjoin, which pads dangling tuples from both of its arguments, as a *full* outerjoin. The syntax is unsurprising:

```
MovieStar NATURAL FULL OUTER JOIN MovieExec;
```

The result of this operation is a relation with the same six-attribute schema as Example 6.24. The tuples of this relation are of three kinds. Those representing individuals who are both stars and executives have tuples with all six attributes non-`NULL`. These are the tuples that are also in the result of Example 6.24.

The second kind of tuple is one for an individual who is a star but not an executive. These tuples have values for attributes `name`, `address`, `gender`, and `birthdate` taken from their tuple in `MovieStar`, while the attributes belonging only to `MovieExec`, namely `cert#` and `netWorth`, have `NULL` values.

The third kind of tuple is for an executive who is not also a star. These tuples have values for the attributes of `MovieExec` taken from their `MovieExec` tuple and `NULL`'s in the attributes `gender` and `birthdate` that come only from `MovieStar`. For instance, the three tuples of the result relation shown in Fig. 6.12 correspond to the three types of individuals, respectively. □

All the variations on the outerjoin that we mentioned in Section 5.2.7 are also available in SQL. If we want a left- or right-outerjoin, we add the appropriate word `LEFT` or `RIGHT` in place of `FULL`. For instance,

```
MovieStar NATURAL LEFT OUTER JOIN MovieExec;
```

name	*address*	*gender*	*birthdate*	*cert#*	*networth*
Mary Tyler Moore	Maple St.	'F'	9/9/99	12345	$100···
Tom Hanks	Cherry Ln.	'M'	8/8/88	NULL	NULL
George Lucas	Oak Rd.	NULL	NULL	23456	$200···

Figure 6.12: Three tuples in the outerjoin of MovieStar and MovieExec

would yield the first two tuples of Fig. 6.12 but not the third. Similarly,

```
MovieStar NATURAL RIGHT OUTER JOIN MovieExec;
```

would yield the first and third tuples of Fig. 6.12 but not the second.

Next, suppose we want a theta-outerjoin instead of a natural outerjoin. Instead of using the keyword NATURAL, we may follow the join by ON and a condition that matching tuples must obey. If we also specify FULL OUTER JOIN, then after matching tuples from the two joined relations, we pad dangling tuples of either relation with NULL's and include the padded tuples in the result.

Example 6.26: Let us reconsider Example 6.23, where we joined the relations Movies and StarsIn using the conditions that the title and movieTitle attributes of the two relations agree and that the year and movieYear attributes of the two relations agree. If we modify that example to call for a full outerjoin:

```
Movies FULL OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

then we shall get not only tuples for movies that have at least one star mentioned in StarsIn, but we shall get tuples for movies with no listed stars, padded with NULL's in attributes movieTitle, movieYear, and starName. Likewise, for stars not appearing in any movie listed in relation Movies we get a tuple with NULL's in the six attributes of Movies. □

The keyword FULL can be replaced by either LEFT or RIGHT in outerjoins of the type suggested by Example 6.26. For instance,

```
Movies LEFT OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

gives us the Movies tuples with at least one listed star and NULL-padded Movies tuples without a listed star, but will not include stars without a listed movie. Conversely,

```
Movies RIGHT OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

will omit the tuples for movies without a listed star but will include tuples for stars not in any listed movies, padded with NULL's.

6.3.9 Exercises for Section 6.3

Exercise 6.3.1: Write the following queries, based on the database schema

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

of Exercise 2.4.1. You should use at least one subquery in each of your answers and write each query in two significantly different ways (e.g., using different sets of the operators EXISTS, IN, ALL, and ANY).

a) Find the makers of PC's with a speed of at least 3.0.

b) Find the printers with the highest price.

! c) Find the laptops whose speed is slower than that of any PC.

! d) Find the model number of the item (PC, laptop, or printer) with the highest price.

! e) Find the maker of the color printer with the lowest price.

!! f) Find the maker(s) of the PC(s) with the fastest processor among all those PC's that have the smallest amount of RAM.

Exercise 6.3.2: Write the following queries, based on the database schema

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

of Exercise 2.4.3. You should use at least one subquery in each of your answers and write each query in two significantly different ways (e.g., using different sets of the operators EXISTS, IN, ALL, and ANY).

a) Find the countries whose ships had the largest number of guns.

! b) Find the classes of ships, at least one of which was sunk in a battle.

c) Find the names of the ships with a 16-inch bore.

d) Find the battles in which ships of the Kongo class participated.

!! e) Find the names of the ships whose number of guns was the largest for those ships of the same bore.

! **Exercise 6.3.3:** Write the query of Fig. 6.10 without any subqueries.

! **Exercise 6.3.4:** Consider expression $\pi_L(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n)$ of relational algebra, where L is a list of attributes all of which belong to R_1. Show that this expression can be written in SQL using subqueries only. More precisely, write an equivalent SQL expression where no `FROM` clause has more than one relation in its list.

! **Exercise 6.3.5:** Write the following queries without using the intersection or difference operators:

a) The intersection query of Fig. 6.5.

b) The difference query of Example 6.17.

!! **Exercise 6.3.6:** We have noticed that certain operators of SQL are redundant, in the sense that they always can be replaced by other operators. For example, we saw that s `IN` R can be replaced by s `= ANY` R. Show that `EXISTS` and `NOT EXISTS` are redundant by explaining how to replace any expression of the form `EXISTS` R or `NOT EXISTS` R by an expression that does not involve `EXISTS` (except perhaps in the expression R itself). *Hint*: Remember that it is permissible to have a constant in the `SELECT` clause.

Exercise 6.3.7: For these relations from our running movie database schema

```
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

describe the tuples that would appear in the following SQL expressions:

a) `Studio CROSS JOIN MovieExec;`

b) `StarsIn NATURAL FULL OUTER JOIN MovieStar;`

c) `StarsIn FULL OUTER JOIN MovieStar ON name = starName;`

! **Exercise 6.3.8:** Using the database schema

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

write a SQL query that will produce information about all products — PC's, laptops, and printers — including their manufacturer if available, and whatever information about that product is relevant (i.e., found in the relation for that type of product).

Exercise 6.3.9: Using the two relations

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
```

from our database schema of Exercise 2.4.3, write a SQL query that will produce all available information about ships, including that information available in the `Classes` relation. You need not produce information about classes if there are no ships of that class mentioned in `Ships`.

! **Exercise 6.3.10:** Repeat Exercise 6.3.9, but also include in the result, for any class C that is not mentioned in `Ships`, information about the ship that has the same name C as its class. You may assume that there is a ship with the class name, even if it doesn't appear in `Ships`.

! **Exercise 6.3.11:** The join operators (other than outerjoin) we learned in this section are redundant, in the sense that they can always be replaced by select-from-where expressions. Explain how to write expressions of the following forms using select-from-where:

a) `R CROSS JOIN S;`

b) `R NATURAL JOIN S;`

c) `R JOIN S ON` C`;`, where C is a SQL condition.

6.4 Full-Relation Operations

In this section we shall study some operations that act on relations as a whole, rather than on tuples individually or in small numbers (as do joins of several relations, for instance). First, we deal with the fact that SQL uses relations that are bags rather than sets, and a tuple can appear more than once in a relation. We shall see how to force the result of an operation to be a set in Section 6.4.1, and in Section 6.4.2 we shall see that it is also possible to prevent the elimination of duplicates in circumstances where SQL systems would normally eliminate them.

Then, we discuss how SQL supports the grouping and aggregation operator γ that we introduced in Section 5.2.4. SQL has aggregation operators and a `GROUP-BY` clause. There is also a "`HAVING`" clause that allows selection of certain groups in a way that depends on the group as a whole, rather than on individual tuples.

6.4.1 Eliminating Duplicates

As mentioned in Section 6.3.4, SQL's notion of relations differs from the abstract notion of relations presented in Section 2.2. A relation, being a set, cannot have more than one copy of any given tuple. When a SQL query creates a new relation, the SQL system does not ordinarily eliminate duplicates. Thus, the SQL response to a query may list the same tuple several times.

Recall from Section 6.2.4 that one of several equivalent definitions of the meaning of a SQL select-from-where query is that we begin with the Cartesian product of the relations referred to in the FROM clause. Each tuple of the product is tested by the condition in the WHERE clause, and the ones that pass the test are given to the output for projection according to the SELECT clause. This projection may cause the same tuple to result from different tuples of the product, and if so, each copy of the resulting tuple is printed in its turn. Further, since there is nothing wrong with a SQL relation having duplicates, the relations from which the Cartesian product is formed may have duplicates, and each identical copy is paired with the tuples from the other relations, yielding a proliferation of duplicates in the product.

If we do not wish duplicates in the result, then we may follow the keyword SELECT by the keyword DISTINCT. That word tells SQL to produce only one copy of any tuple and is the SQL analog of applying the δ operator of Section 5.2.1 to the result of the query.

Example 6.27: Let us reconsider the query of Fig. 6.9, where we asked for the producers of Harrison Ford's movies using no subqueries. As written, George Lucas will appear many times in the output. If we want only to see each producer once, we may change line (1) of the query to

```
1)  SELECT DISTINCT name
```

Then, the list of producers will have duplicate occurrences of names eliminated before printing.

Incidentally, the query of Fig. 6.7, where we used subqueries, does not necessarily suffer from the problem of duplicate answers. True, the subquery at line (4) of Fig. 6.7 will produce the certificate number of George Lucas several times. However, in the "main" query of line (1), we examine each tuple of MovieExec once. Presumably, there is only one tuple for George Lucas in that relation, and if so, it is only this tuple that satisfies the WHERE clause of line (3). Thus, George Lucas is printed only once. □

6.4.2 Duplicates in Unions, Intersections, and Differences

Unlike the SELECT statement, which preserves duplicates as a default and only eliminates them when instructed to by the DISTINCT keyword, the union, intersection, and difference operations, which we introduced in Section 6.2.5, normally eliminate duplicates. That is, bags are converted to sets, and the set version of the operation is applied. In order to prevent the elimination of duplicates, we must follow the operator UNION, INTERSECT, or EXCEPT by the keyword ALL. If we do, then we get the bag semantics of these operators as was discussed in Section 5.1.2.

Example 6.28: Consider again the union expression from Example 6.18, but now add the keyword ALL, as:

The Cost of Duplicate Elimination

One might be tempted to place DISTINCT after every SELECT, on the theory that it is harmless. In fact, it is very expensive to eliminate duplicates from a relation. The relation must be sorted or partitioned so that identical tuples appear next to each other. Only by grouping the tuples in this way can we determine whether or not a given tuple should be eliminated. The time it takes to sort the relation so that duplicates may be eliminated is often greater than the time it takes to execute the query itself. Thus, duplicate elimination should be used judiciously if we want our queries to run fast.

```
(SELECT title, year FROM Movies)
    UNION ALL
(SELECT movieTitle AS title, movieYear AS year FROM StarsIn);
```

Now, a title and year will appear as many times in the result as it appears in each of the relations `Movies` and `StarsIn` put together. For instance, if a movie appeared once in the `Movies` relation and there were three stars for that movie listed in `StarsIn` (so the movie appeared in three different tuples of `StarsIn`), then that movie's title and year would appear four times in the result of the union. □

As for union, the operators `INTERSECT ALL` and `EXCEPT ALL` are intersection and difference of bags. Thus, if R and S are relations, then the result of expression

$$R \texttt{ INTERSECT ALL } S$$

is the relation in which the number of times a tuple t appears is the minimum of the number of times it appears in R and the number of times it appears in S.

The result of expression

$$R \texttt{ EXCEPT ALL } S$$

has tuple t as many times as the difference of the number of times it appears in R minus the number of times it appears in S, provided the difference is positive. Each of these definitions is what we discussed for bags in Section 5.1.2.

6.4.3 Grouping and Aggregation in SQL

In Section 5.2.4, we introduced the grouping-and-aggregation operator γ for our extended relational algebra. Recall that this operator allows us to partition the tuples of a relation into "groups," based on the values of tuples in one or more attributes, as discussed in Section 5.2.3. We are then able to aggregate certain other columns of the relation by applying "aggregation" operators to those columns. If there are groups, then the aggregation is done separately for each group. SQL provides all the capability of the γ operator through the use of aggregation operators in SELECT clauses and a special GROUP BY clause.

6.4.4 Aggregation Operators

SQL uses the five aggregation operators SUM, AVG, MIN, MAX, and COUNT that we met in Section 5.2.2. These operators are used by applying them to a scalar-valued expression, typically a column name, in a SELECT clause. One exception is the expression COUNT(*), which counts all the tuples in the relation that is constructed from the FROM clause and WHERE clause of the query.

In addition, we have the option of eliminating duplicates from the column before applying the aggregation operator by using the keyword DISTINCT. That is, an expression such as COUNT(DISTINCT x) counts the number of distinct values in column x. We could use any of the other operators in place of COUNT here, but expressions such as SUM(DISTINCT x) rarely make sense, since it asks us to sum the different values in column x.

Example 6.29: The following query finds the average net worth of all movie executives:

```
SELECT AVG(netWorth)
FROM MovieExec;
```

Note that there is no WHERE clause at all, so the keyword WHERE is properly omitted. This query examines the netWorth column of the relation

```
MovieExec(name, address, cert#, netWorth)
```

sums the values found there, one value for each tuple (even if the tuple is a duplicate of some other tuple), and divides the sum by the number of tuples. If there are no duplicate tuples, then this query gives the average net worth as we expect. If there were duplicate tuples, then a movie executive whose tuple appeared n times would have his or her net worth counted n times in the average. □

Example 6.30: The following query:

```
SELECT COUNT(*)
FROM StarsIn;
```

counts the number of tuples in the StarsIn relation. The similar query:

```
SELECT COUNT(starName)
FROM StarsIn;
```

counts the number of values in the starName column of the relation. Since duplicate values are not eliminated when we project onto the starName column in SQL, this count should be the same as the count produced by the query with COUNT(*).

If we want to be certain that we do not count duplicate values more than once, we can use the keyword DISTINCT before the aggregated attribute, as:

```
SELECT COUNT(DISTINCT starName)
FROM StarsIn;
```

Now, each star is counted once, no matter in how many movies they appeared. □

6.4.5 Grouping

To group tuples, we use a GROUP BY clause, following the WHERE clause. The keywords GROUP BY are followed by a list of *grouping* attributes. In the simplest situation, there is only one relation reference in the FROM clause, and this relation has its tuples grouped according to their values in the grouping attributes. Whatever aggregation operators are used in the SELECT clause are applied only within groups.

Example 6.31: The problem of finding, from the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

the sum of the lengths of all movies for each studio is expressed by

```
SELECT studioName, SUM(length)
FROM Movies
GROUP BY studioName;
```

We may imagine that the tuples of relation Movies are reorganized and grouped so that all the tuples for Disney studios are together, all those for MGM are together, and so on, as was suggested in Fig. 5.4. The sums of the length components of all the tuples in each group are calculated, and for each group, the studio name is printed along with that sum. □

Observe in Example 6.31 how the SELECT clause has two kinds of terms. These are the only terms that may appear when there is an aggregation in the SELECT clause.

1. Aggregations, where an aggregate operator is applied to an attribute or expression involving attributes. As mentioned, these terms are evaluated on a per-group basis.

2. Attributes, such as studioName in this example, that appear in the GROUP BY clause. In a SELECT clause that has aggregations, only those attributes that are mentioned in the GROUP BY clause may appear unaggregated in the SELECT clause.

While queries involving GROUP BY generally have both grouping attributes and aggregations in the SELECT clause, it is technically not necessary to have both. For example, we could write

```
SELECT studioName
FROM Movies
GROUP BY studioName;
```

This query would group the tuples of Movies according to their studio name and then print the studio name for each group, no matter how many tuples there are with a given studio name. Thus, the above query has the same effect as

```
SELECT DISTINCT studioName
FROM Movies;
```

It is also possible to use a GROUP BY clause in a query about several relations. Such a query is interpreted by the following sequence of steps:

1. Evaluate the relation R expressed by the FROM and WHERE clauses. That is, relation R is the Cartesian product of the relations mentioned in the FROM clause, to which the selection of the WHERE clause is applied.

2. Group the tuples of R according to the attributes in the GROUP BY clause.

3. Produce as a result the attributes and aggregations of the SELECT clause, as if the query were about a stored relation R.

Example 6.32: Suppose we wish to print a table listing each producer's total length of film produced. We need to get information from the two relations

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

so we begin by taking their theta-join, equating the certificate numbers from the two relations. That step gives us a relation in which each MovieExec tuple is paired with the Movies tuples for all the movies of that producer. Note that an executive who is not a producer will not be paired with any movies, and therefore will not appear in the relation. Now, we can group the selected tuples of this relation according to the name of the producer. Finally, we sum the lengths of the movies in each group. The query is shown in Fig. 6.13. □

```
SELECT name, SUM(length)
FROM MovieExec, Movies
WHERE producerC# = cert#
GROUP BY name;
```

Figure 6.13: Computing the length of movies for each producer

6.4.6 Grouping, Aggregation, and Nulls

When tuples have nulls, there are a few rules we must remember:

- The value NULL is ignored in any aggregation. It does not contribute to a sum, average, or count of an attribute, nor can it be the minimum or maximum in its column. For example, COUNT(*) is always a count of the number of tuples in a relation, but COUNT(A) is the number of tuples with non-NULL values for attribute A.

- On the other hand, NULL is treated as an ordinary value when forming groups. That is, we can have a group in which one or more of the grouping attributes are assigned the value NULL.

- When we perform any aggregation except count over an empty bag of values, the result is NULL. The count of an empty bag is 0.

Example 6.33: Suppose we have a relation $R(A, B)$ with one tuple, both of whose components are NULL:

A	B
NULL	NULL

Then the result of:

```
SELECT A, COUNT(B)
FROM R
GROUP BY A;
```

is the one tuple (NULL, 0). The reason is that when we group by A, we find only a group for value NULL. This group has one tuple, and its B-value is NULL. We thus count the bag of values {NULL}. Since the count of a bag of values does not count the NULL's, this count is 0.

On the other hand, the result of:

```
SELECT A, SUM(B)
FROM R
GROUP BY A;
```

Order of Clauses in SQL Queries

We have now met all six clauses that can appear in a SQL "select-from-where" query: `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY`. Only the `SELECT` and `FROM` clauses are required. Whichever additional clauses appear must be in the order listed above.

is the one tuple (`NULL,NULL`). The reason is as follows. The group for value `NULL` has one tuple, the only tuple in R. However, when we try to sum the B-values for this group, we only find `NULL`, and `NULL` does not contribute to a sum. Thus, we are summing an empty bag of values, and this sum is defined to be `NULL`. □

6.4.7 HAVING Clauses

Suppose that we did not wish to include all of the producers in our table of Example 6.32. We could restrict the tuples prior to grouping in a way that would make undesired groups empty. For instance, if we only wanted the total length of movies for producers with a net worth of more than \$10,000,000, we could change the third line of Fig. 6.13 to

```
WHERE producerC# = cert# AND networth > 10000000
```

However, sometimes we want to choose our groups based on some aggregate property of the group itself. Then we follow the `GROUP BY` clause with a `HAVING` clause. The latter clause consists of the keyword `HAVING` followed by a condition about the group.

Example 6.34: Suppose we want to print the total film length for only those producers who made at least one film prior to 1930. We may append to Fig. 6.13 the clause

```
HAVING MIN(year) < 1930
```

The resulting query, shown in Fig. 6.14, would remove from the grouped relation all those groups in which every tuple had a `year` component 1930 or higher. □

There are several rules we must remember about `HAVING` clauses:

- An aggregation in a `HAVING` clause applies only to the tuples of the group being tested.
- Any attribute of relations in the `FROM` clause may be aggregated in the `HAVING` clause, but only those attributes that are in the `GROUP BY` list may appear unaggregated in the `HAVING` clause (the same rule as for the `SELECT` clause).

```
SELECT name, SUM(length)
FROM MovieExec, Movies
WHERE producerC# = cert#
GROUP BY name
HAVING MIN(year) < 1930;
```

Figure 6.14: Computing the total length of film for early producers

6.4.8 Exercises for Section 6.4

Exercise 6.4.1: Write each of the queries in Exercise 2.4.1 in SQL, making sure that duplicates are eliminated.

Exercise 6.4.2: Write each of the queries in Exercise 2.4.3 in SQL, making sure that duplicates are eliminated.

! **Exercise 6.4.3:** For each of your answers to Exercise 6.3.1, determine whether or not the result of your query can have duplicates. If so, rewrite the query to eliminate duplicates. If not, write a query without subqueries that has the same, duplicate-free answer.

! **Exercise 6.4.4:** Repeat Exercise 6.4.3 for your answers to Exercise 6.3.2.

! **Exercise 6.4.5:** In Example 6.27, we mentioned that different versions of the query "find the producers of Harrison Ford's movies" can have different answers as bags, even though they yield the same set of answers. Consider the version of the query in Example 6.22, where we used a subquery in the `FROM` clause. Does this version produce duplicates, and if so, why?

Exercise 6.4.6: Write the following queries, based on the database schema

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

of Exercise 2.4.1, and evaluate your queries using the data of that exercise.

a) Find the average speed of PC's.

b) Find the average speed of laptops costing over $1000.

c) Find the average price of PC's made by manufacturer "A."

! d) Find the average price of PC's and laptops made by manufacturer "D."

e) Find, for each different speed, the average price of a PC.

! f) Find for each manufacturer, the average screen size of its laptops.

! g) Find the manufacturers that make at least three different models of PC.

! h) Find for each manufacturer who sells PC's the maximum price of a PC.

! i) Find, for each speed of PC above 2.0, the average price.

!! j) Find the average hard disk size of a PC for all those manufacturers that make printers.

Exercise 6.4.7: Write the following queries, based on the database schema

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

of Exercise 2.4.3, and evaluate your queries using the data of that exercise.

a) Find the number of battleship classes.

b) Find the average number of guns of battleship classes.

! c) Find the average number of guns of battleships. Note the difference between (b) and (c); do we weight a class by the number of ships of that class or not?

! d) Find for each class the year in which the first ship of that class was launched.

! e) Find for each class the number of ships of that class sunk in battle.

!! f) Find for each class with at least three ships the number of ships of that class sunk in battle.

!! g) The weight (in pounds) of the shell fired from a naval gun is approximately one half the cube of the bore (in inches). Find the average weight of the shell for each country's ships.

Exercise 6.4.8: In Example 5.10 we gave an example of the query: "find, for each star who has appeared in at least three movies, the earliest year in which they appeared." We wrote this query as a γ operation. Write it in SQL.

! **Exercise 6.4.9:** The γ operator of extended relational algebra does not have a feature that corresponds to the HAVING clause of SQL. Is it possible to mimic a SQL query with a HAVING clause in relational algebra? If so, how would we do it in general?

6.5 Database Modifications

To this point, we have focused on the normal SQL query form: the select-from-where statement. There are a number of other statement forms that do not return a result, but rather change the state of the database. In this section, we shall focus on three types of statements that allow us to

1. Insert tuples into a relation.

2. Delete certain tuples from a relation.

3. Update values of certain components of certain existing tuples.

We refer to these three types of operations collectively as *modifications*.

6.5.1 Insertion

The basic form of insertion statement is:

INSERT INTO $R(A_1, \ldots, A_n)$ VALUES $(v_1, \ldots, v_n)$;

A tuple is created using the value v_i for attribute A_i, for $i = 1, 2, \ldots, n$. If the list of attributes does not include all attributes of the relation R, then the tuple created has default values for all missing attributes.

Example 6.35: Suppose we wish to add Sydney Greenstreet to the list of stars of *The Maltese Falcon*. We say:

```
1) INSERT INTO StarsIn(movieTitle, movieYear, starName)
2) VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');
```

The effect of executing this statement is that a tuple with the three components on line (2) is inserted into the relation `StarsIn`. Since all attributes of `StarsIn` are mentioned on line (1), there is no need to add default components. The values on line (2) are matched with the attributes on line (1) in the order given, so `'The Maltese Falcon'` becomes the value of the component for attribute `movieTitle`, and so on. □

If, as in Example 6.35, we provide values for all attributes of the relation, then we may omit the list of attributes that follows the relation name. That is, we could just say:

```
INSERT INTO StarsIn
VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');
```

However, if we take this option, we must be sure that the order of the values is the same as the standard order of attributes for the relation.

- If you are not sure of the declared order for the attributes, it is best to list them in the INSERT clause in the order you choose for their values in the VALUES clause.

The simple INSERT described above only puts one tuple into a relation. Instead of using explicit values for one tuple, we can compute a set of tuples to be inserted, using a subquery. This subquery replaces the keyword VALUES and the tuple expression in the INSERT statement form described above.

Example 6.36: Suppose we want to add to the relation

```
Studio(name, address, presC#)
```

all movie studios that are mentioned in the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

but do not appear in Studio. Since there is no way to determine an address or a president for such a studio, we shall have to be content with value NULL for attributes address and presC# in the inserted Studio tuples. A way to make this insertion is shown in Fig. 6.15.

```
1)  INSERT INTO Studio(name)
2)      SELECT DISTINCT studioName
3)      FROM Movies
4)      WHERE studioName NOT IN
5)          (SELECT name
6)           FROM Studio);
```

Figure 6.15: Adding new studios

Like most SQL statements with nesting, Fig. 6.15 is easiest to examine from the inside out. Lines (5) and (6) generate all the studio names in the relation Studio. Thus, line (4) tests that a studio name from the Movies relation is none of these studios.

Now, we see that lines (2) through (6) produce the set of studio names found in Movies but not in Studio. The use of DISTINCT on line (2) assures that each studio will appear only once in this set, no matter how many movies it owns. Finally, line (1) inserts each of these studios, with NULL for the attributes address and presC#, into relation Studio. □

6.5.2 Deletion

The form of a deletion is

DELETE FROM R WHERE <condition>;

The Timing of Insertions

The SQL standard requires that the query be evaluated completely before any tuples are inserted. For example, in Fig. 6.15, the query of lines (2) through (6) must be evaluated prior to executing the insertion of line (1). Thus, there is no possibility that new tuples added to `Studio` at line (1) will affect the condition on line (4).

In this particular example, it does not matter whether or not insertions are delayed until the query is completely evaluated. However, suppose `DISTINCT` were removed from line (2) of Fig. 6.15. If we evaluate the query of lines (2) through (6) before doing any insertion, then a new studio name appearing in several `Movies` tuples would appear several times in the result of this query and therefore would be inserted several times into relation `Studio`. However, if the DBMS inserted new studios into `Studio` as soon as we found them during the evaluation of the query of lines (2) through (6), something that would be incorrect according to the standard, then the same new studio would not be inserted twice. Rather, as soon as the new studio was inserted once, its name would no longer satisfy the condition of lines (4) through (6), and it would not appear a second time in the result of the query of lines (2) through (6).

The effect of executing this statement is that every tuple satisfying the condition will be deleted from relation R.

Example 6.37: We can delete from relation

```
StarsIn(movieTitle, movieYear, starName)
```

the fact that Sydney Greenstreet was a star in *The Maltese Falcon* by the SQL statement:

```
DELETE FROM StarsIn
WHERE movieTitle = 'The Maltese Falcon' AND
      movieYear = 1942 AND
      starName = 'Sydney Greenstreet';
```

Notice that unlike the insertion statement of Example 6.35, we cannot simply specify a tuple to be deleted. Rather, we must describe the tuple exactly by a `WHERE` clause. □

Example 6.38: Here is another example of a deletion. This time, we delete from relation

```
MovieExec(name, address, cert#, netWorth)
```

several tuples at once by using a condition that can be satisfied by more than one tuple. The statement

```
DELETE FROM MovieExec
WHERE netWorth < 10000000;
```

deletes all movie executives whose net worth is low — less than ten million dollars. □

6.5.3 Updates

While we might think of both insertions and deletions of tuples as "updates" to the database, an *update* in SQL is a very specific kind of change to the database: one or more tuples that already exist in the database have some of their components changed. The general form of an update statement is:

UPDATE *R* SET <new-value assignments> WHERE <condition>;

Each new-value assignment is an attribute, an equal sign, and an expression. If there is more than one assignment, they are separated by commas. The effect of this statement is to find all the tuples in *R* that satisfy the condition. Each of these tuples is then changed by having the expressions in the assignments evaluated and assigned to the components of the tuple for the corresponding attributes of *R*.

Example 6.39: Let us modify the relation

```
MovieExec(name, address, cert#, netWorth)
```

by attaching the title Pres. in front of the name of every movie executive who is the president of a studio. The condition the desired tuples satisfy is that their certificate numbers appear in the presC# component of some tuple in the Studio relation. We express this update as:

```
1) UPDATE MovieExec
2) SET name = 'Pres. ' || name
3) WHERE cert# IN (SELECT presC# FROM Studio);
```

Line (3) tests whether the certificate number from the MovieExec tuple is one of those that appear as a president's certificate number in Studio.

Line (2) performs the update on the selected tuples. Recall that the operator || denotes concatenation of strings, so the expression following the = sign in line (2) places the characters Pres. and a blank in front of the old value of the name component of this tuple. The new string becomes the value of the name component of this tuple; the effect is that 'Pres. ' has been prepended to the old value of name. □

6.5.4 Exercises for Section 6.5

Exercise 6.5.1: Write the following database modifications, based on the database schema

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

of Exercise 2.4.1. Describe the effect of the modifications on the data of that exercise.

a) Using two INSERT statements, store in the database the fact that PC model 1100 is made by manufacturer C, has speed 3.2, RAM 1024, hard disk 180, and sells for $2499.

! b) Insert the facts that for every PC there is a laptop with the same manufacturer, speed, RAM, and hard disk, a 17-inch screen, a model number 1100 greater, and a price $500 more.

c) Delete all PC's with less than 100 gigabytes of hard disk.

d) Delete all laptops made by a manufacturer that doesn't make printers.

e) Manufacturer A buys manufacturer B. Change all products made by B so they are now made by A.

f) For each PC, double the amount of RAM and add 60 gigabytes to the amount of hard disk. (Remember that several attributes can be changed by one UPDATE statement.)

! g) For each laptop made by manufacturer B, add one inch to the screen size and subtract $100 from the price.

Exercise 6.5.2: Write the following database modifications, based on the database schema

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

of Exercise 2.4.3. Describe the effect of the modifications on the data of that exercise.

a) The two British battleships of the Nelson class — Nelson and Rodney — were both launched in 1927, had nine 16-inch guns, and a displacement of 34,000 tons. Insert these facts into the database.

b) Two of the three battleships of the Italian Vittorio Veneto class — Vittorio Veneto and Italia — were launched in 1940; the third ship of that class, Roma, was launched in 1942. Each had nine 15-inch guns and a displacement of 41,000 tons. Insert these facts into the database.

c) Delete from `Ships` all ships sunk in battle.

d) Modify the `Classes` relation so that gun bores are measured in centimeters (one inch = 2.5 centimeters) and displacements are measured in metric tons (one metric ton = 1.1 tons).

e) Delete all classes with fewer than three ships.

6.6 Transactions in SQL

To this point, our model of operations on the database has been that of one user querying or modifying the database. Thus, operations on the database are executed one at a time, and the database state left by one operation is the state upon which the next operation acts. Moreover, we imagine that operations are carried out in their entirety ("atomically"). That is, we assumed it is impossible for the hardware or software to fail in the middle of a modification, leaving the database in a state that cannot be explained as the result of the operations performed on it.

Real life is often considerably more complicated. We shall first consider what can happen to leave the database in a state that doesn't reflect the operations performed on it, and then we shall consider the tools SQL gives the user to assure that these problems do not occur.

6.6.1 Serializability

In applications like Web services, banking, or airline reservations, hundreds of operations per second may be performed on the database. The operations initiate at any of thousands or millions of sites, such as desktop computers or automatic teller machines. It is entirely possible that we could have two operations affecting the same bank account or flight, and for those operations to overlap in time. If so, they might interact in strange ways.

Here is an example of what could go wrong if the DBMS were completely unconstrained as to the order in which it operated upon the database. This example involves a database interacting with people, and it is intended to illustrate why it is important to control the sequences in which interacting events can occur. However, a DBMS would not control events that were so "large" that they involved waiting for a user to make a choice. The event sequences controlled by the DBMS involve only the execution of SQL statements.

Example 6.40: The typical airline gives customers a Web interface where they can choose a seat for their flight. This interface shows a map of available seats, and the data for this map is obtained from the airline's database. There might be a relation such as:

```
Flights(fltNo, fltDate, seatNo, seatStatus)
```

upon which we can issue the query:

```
SELECT seatNo
FROM Flights
WHERE fltNo = 123 AND fltDate = DATE '2008-12-25'
    AND seatStatus = 'available';
```

The flight number and date are example data, which would in fact be obtained from previous interactions with the customer.

When the customer clicks on an empty seat, say 22A, that seat is reserved for them. The database is modified by an update-statement, such as:

```
UPDATE Flights
SET seatStatus = 'occupied'
WHERE fltNo = 123 AND fltDate = DATE '2008-12-25'
    AND seatNo = '22A';
```

However, this customer may not be the only one reserving a seat on flight 123 on Dec. 25, 2008 and this exact moment. Another customer may have asked for the seat map at the same time, in which case they also see seat 22A empty. Should they also choose seat 22A, they too believe they have reserved 22A. The timing of these events is as suggested by Fig. 6.16. □

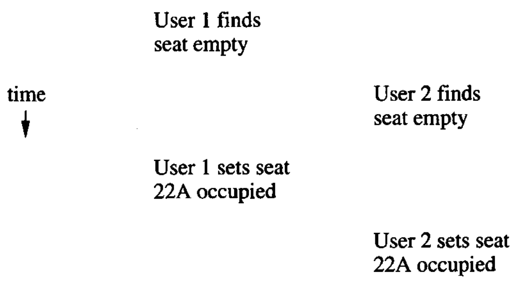

Figure 6.16: Two customers trying to book the same seat simultaneously

As we see from Example 6.40, it is conceivable that two operations could each be performed correctly, and yet the global result not be correct: both customers believe they have been granted seat 22A. The problem is solved in SQL by the notion of a "transaction," which is informally a group of operations that need to be performed together. Suppose that in Example 6.40, the query

Assuring Serializable Behavior

In practice it is often impossible to require that operations run serially; there are just too many of them, and some parallelism is required. Thus, DBMS's adopt a mechanism for assuring serializable behavior; even if the execution is not serial, the result looks to users as if operations were executed serially.

One common approach is for the DBMS to *lock* elements of the database so that two functions cannot access them at the same time. We mentioned locking in Section 1.2.4, and there is an extensive technology of how to implement locks in a DBMS. For example, if the transaction of Example 6.40 were written to lock other transactions out of the `Flights` relation, then transactions that did not access `Flights` could run in parallel with the seat-selection transaction, but no other invocation of the seat-selection operation could run in parallel.

and update shown would be grouped into one transaction.[6] SQL then allows the programmer to state that a certain transaction must be *serializable* with respect to other transactions. That is, these transactions must behave as if they were run *serially* — one at a time, with no overlap.

Clearly, if the two invocations of the seat-selection operation are run serially (or serializably), then the error we saw cannot occur. One customer's invocation occurs first. This customer sees seat 22A is empty, and books it. The other customer's invocation then begins and is not given 22A as a choice, because it is already occupied. It may matter to the customers who gets the seat, but to the database all that is important is that a seat is assigned only once.

6.6.2 Atomicity

In addition to nonserialized behavior that can occur if two or more database operations are performed about the same time, it is possible for a single operation to put the database in an unacceptable state if there is a hardware or software "crash" while the operation is executing. Here is another example suggesting what might occur. As in Example 6.40, we should remember that real database systems do not allow this sort of error to occur in properly designed application programs.

Example 6.41: Let us picture another common sort of database: a bank's account records. We can represent the situation by a relation

[6]However, it would be extremely unwise to group into a single transaction operations that involved a user, or even a computer that was not owned by the airline, such as a travel agent's computer. Another mechanism must be used to deal with event sequences that include operations outside the database.

```
Accounts(acctNo, balance)
```

Consider the operation of transferring $100 from the account numbered 123 to the account 456. We might first check whether there is at least $100 in account 123, and if so, we execute the following two steps:

1. Add $100 to account 456 by the SQL update statement:

```
UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;
```

2. Subtract $100 from account 123 by the SQL update statement:

```
UPDATE Accounts
SET balance = balance - 100
WHERE acctNo = 123;
```

Now, consider what happens if there is a failure after Step (1) but before Step (2). Perhaps the computer fails, or the network connecting the database to the processor that is actually performing the transfer fails. Then the database is left in a state where money has been transferred into the second account, but the money has not been taken out of the first account. The bank has in effect given away the amount of money that was to be transferred. □

The problem illustrated by Example 6.41 is that certain combinations of database operations, like the two updates of that example, need to be done *atomically*; that is, either they are both done or neither is done. For example, a simple solution is to have all changes to the database done in a local workspace, and only after all work is done do we *commit* the changes to the database, whereupon all changes become part of the database and visible to other operations.

6.6.3 Transactions

The solution to the problems of serialization and atomicity posed in Sections 6.6.1 and 6.6.2 is to group database operations into *transactions*. A transaction is a collection of one or more operations on the database that must be executed atomically; that is, either all operations are performed or none are. In addition, SQL requires that, as a default, transactions are executed in a serializable manner. A DBMS may allow the user to specify a less stringent constraint on the interleaving of operations from two or more transactions. We shall discuss these modifications to the serializability condition in later sections.

When using the *generic SQL interface* (the facility wherein one types queries and other SQL statements), each statement is a transaction by itself. However,

How the Database Changes During Transactions

Different systems may do different things to implement transactions. It is possible that as a transaction executes, it makes changes to the database. If the transaction aborts, then (unless the programmer took precautions) it is possible that these changes were seen by some other transaction. The most common solution is for the database system to lock the changed items until `COMMIT` or `ROLLBACK` is chosen, thus preventing other transactions from seeing the tentative change. Locks or an equivalent would surely be used if the user wants the transactions to run in a serializable fashion.

However, as we shall see starting in Section 6.6.4, SQL offers us several options regarding the treatment of tentative database changes. It is possible that the changed data is not locked and becomes visible even though a subsequent rollback makes the change disappear. It is up to the author of a transaction to decide whether it is safe for that transaction to see tentative changes of other transactions.

SQL allows the programmer to group several statements into a single transaction. The SQL command `START TRANSACTION` is used to mark the beginning of a transaction. There are two ways to end a transaction:

1. The SQL statement `COMMIT` causes the transaction to end successfully. Whatever changes to the database were caused by the SQL statement or statements since the current transaction began are installed permanently in the database (i.e., they are *committed*). Before the `COMMIT` statement is executed, changes are tentative and may or may not be visible to other transactions.

2. The SQL statement `ROLLBACK` causes the transaction to *abort*, or terminate unsuccessfully. Any changes made in response to the SQL statements of the transaction are undone (i.e., they are *rolled back*), so they never permanently appear in the database.

Example 6.42: Suppose we want the transfer operation of Example 6.41 to be a single transaction. We execute `BEGIN TRANSACTION` before accessing the database. If we find that there are insufficient funds to make the transfer, then we would execute the `ROLLBACK` command. However, if there are sufficient funds, then we execute the two update statements and then execute `COMMIT`. □

6.6.4 Read-Only Transactions

Examples 6.40 and 6.41 each involved a transaction that read and then (possibly) wrote some data into the database. This sort of transaction is prone to

Application- Versus System-Generated Rollbacks

In our discussion of transactions, we have presumed that the decision whether a transaction is committed or rolled back is made as part of the application issuing the transaction. That is, as in Examples 6.44 and 6.42, a transaction may perform a number of database operations, then decide whether to make any changes permanent by issuing `COMMIT`, or to return to the original state by issuing `ROLLBACK`. However, the system may also perform transaction rollbacks, to ensure that transactions are executed atomically and conform to their specified isolation level in the presence of other concurrent transactions or system crashes. Typically, if the system aborts a transaction then a special error code or exception is generated. If an application wishes to guarantee that its transactions are executed successfully, it must catch such conditions and reissue the transaction in question.

serialization problems. Thus we saw in Example 6.40 what could happen if two executions of the function tried to book the same seat at the same time, and we saw in Example 6.41 what could happen if there was a crash in the middle of a funds transfer. However, when a transaction only reads data and does not write data, we have more freedom to let the transaction execute in parallel with other transactions.

Example 6.43: Suppose we wrote a program that read data from the `Flights` relation of Example 6.40 to determine whether a certain seat was available. We could execute many invocations of this program at once, without risk of permanent harm to the database. The worst that could happen is that while we were reading the availability of a certain seat, that seat was being booked or was being released by the execution of some other program. Thus, we might get the answer "available" or "occupied," depending on microscopic differences in the time at which we executed the query, but the answer would make sense at some time. □

If we tell the SQL execution system that our current transaction is *read-only*, that is, it will never change the database, then it is quite possible that the SQL system will be able to take advantage of that knowledge. Generally it will be possible for many read-only transactions accessing the same data to run in parallel, while they would not be allowed to run in parallel with a transaction that wrote the same data.

We tell the SQL system that the next transaction is read-only by:

```
SET TRANSACTION READ ONLY;
```

This statement must be executed before the transaction begins. We can also inform SQL that the coming transaction may write data by the statement

```
    SET TRANSACTION READ WRITE;
```

However, this option is the default.

6.6.5 Dirty Reads

Dirty data is a common term for data written by a transaction that has not yet committed. A *dirty read* is a read of dirty data written by another transaction. The risk in reading dirty data is that the transaction that wrote it may eventually abort. If so, then the dirty data will be removed from the database, and the world is supposed to behave as if that data never existed. If some other transaction has read the dirty data, then that transaction might commit or take some other action that reflects its knowledge of the dirty data.

Sometimes the dirty read matters, and sometimes it doesn't. Other times it matters little enough that it makes sense to risk an occasional dirty read and thus avoid:

1. The time-consuming work by the DBMS that is needed to prevent dirty reads, and

2. The loss of parallelism that results from waiting until there is no possibility of a dirty read.

Here are some examples of what might happen when dirty reads are allowed.

Example 6.44: Let us reconsider the account transfer of Example 6.41. However, suppose that transfers are implemented by a program P that executes the following sequence of steps:

1. Add money to account 2.

2. Test if account 1 has enough money.

    (a) If there is not enough money, remove the money from account 2 and end.[7]

    (b) If there is enough money, subtract the money from account 1 and end.

If program P is executed serializably, then it doesn't matter that we have put money temporarily into account 2. No one will see that money, and it gets removed if the transfer can't be made.

However, suppose dirty reads are possible. Imagine there are three accounts: $A1$, $A2$, and $A3$, with \$100, \$200, and \$300, respectively. Suppose transaction

[7]You should be aware that the program P is trying to perform functions that would more typically be done by the DBMS. In particular, when P decides, as it has done at this step, that it must not complete the transaction, it would issue a rollback (abort) command to the DBMS and have the DBMS reverse the effects of this execution of P.

T_1 executes program P to transfer \$150 from $A1$ to $A2$. At roughly the same time, transaction T_2 runs program P to transfer \$250 from $A2$ to $A3$. Here is a possible sequence of events:

1. T_2 executes Step (1) and adds \$250 to $A3$, which now has \$550.
2. T_1 executes Step (1) and adds \$150 to $A2$, which now has \$350.
3. T_2 executes the test of Step (2) and finds that $A2$ has enough funds (\$350) to allow the transfer of \$250 from $A2$ to $A3$.
4. T_1 executes the test of Step (2) and finds that $A1$ does not have enough funds (\$100) to allow the transfer of \$150 from $A1$ to $A2$.
5. T_2 executes Step (2b). It subtracts \$250 from $A2$, which now has \$100, and ends.
6. T_1 executes Step (2a). It subtracts \$150 from $A2$, which now has $-$\$50, and ends.

The total amount of money has not changed; there is still \$600 among the three accounts. But because T_2 read dirty data at the third of the six steps above, we have not protected against an account going negative, which supposedly was the purpose of testing the first account to see if it had adequate funds. □

Example 6.45: Let us imagine a variation on the seat-choosing function of Example 6.40. In the new approach:

1. We find an available seat and reserve it by setting `seatStatus` to `'occupied'` for that seat. If there is none, end.
2. We ask the customer for approval of the seat. If so, we commit. If not, we release the seat by setting `seatStatus` to `'available'` and repeat Step (1) to get another seat.

If two transactions are executing this algorithm at about the same time, one might reserve a seat S, which later is rejected by the customer. If the second transaction executes Step (1) at a time when seat S is marked occupied, the customer for that transaction is not given the option to take seat S.

As in Example 6.44, the problem is that a dirty read has occurred. The second transaction saw a tuple (with S marked occupied) that was written by the first transaction and later modified by the first transaction. □

How important is the fact that a read was dirty? In Example 6.44 it was very important; it caused an account to go negative despite apparent safeguards against that happening. In Example 6.45, the problem does not look too serious. Indeed, the second traveler might not get their favorite seat, or might even be told that no seats existed. However, in the latter case, running the transaction again will almost certainly reveal the availability of seat S. It might well make sense to implement this seat-choosing function in a way that allowed dirty reads, in order to speed up the average processing time for booking requests.

SQL allows us to specify that dirty reads are acceptable for a given transaction. We use the SET TRANSACTION statement that we discussed in Section 6.6.4. The appropriate form for a transaction like that described in Example 6.45 is:

```
1)  SET TRANSACTION READ WRITE
2)      ISOLATION LEVEL READ UNCOMMITTED;
```

The statement above does two things:

1. Line (1) declares that the transaction may write data.

2. Line (2) declares that the transaction may run with the "isolation level" *read-uncommitted.* That is, the transaction is allowed to read dirty data. We shall discuss the four isolation levels in Section 6.6.6. So far, we have seen two of them: serializable and read-uncommitted.

Note that if the transaction is not read-only (i.e., it may modify the database), and we specify isolation level READ UNCOMMITTED, then we must also specify READ WRITE. Recall from Section 6.6.4 that the default assumption is that transactions are read-write. However, SQL makes an exception for the case where dirty reads are allowed. Then, the default assumption is that the transaction is read-only, because read-write transactions with dirty reads entail significant risks, as we saw. If we want a read-write transaction to run with read-uncommitted as the isolation level, then we need to specify READ WRITE explicitly, as above.

6.6.6 Other Isolation Levels

SQL provides a total of four *isolation levels.* Two of them we have already seen: serializable and read-uncommitted (dirty reads allowed). The other two are *read-committed* and *repeatable-read.* They can be specified for a given transaction by

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

or

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

respectively. For each, the default is that transactions are read-write, so we can add READ ONLY to either statement, if appropriate. Incidentally, we also have the option of specifying

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Interactions Among Transactions Running at Different Isolation Levels

A subtle point is that the isolation level of a transaction affects only what data *that* transaction may see; it does not affect what any other transaction sees. As a case in point, if a transaction T is running at level serializable, then the execution of T must appear as if all other transactions run either entirely before or entirely after T. However, if some of those transactions are running at another isolation level, then *they* may see the data written by T as T writes it. They may even see dirty data from T if they are running at isolation level read-uncommitted, and T aborts.

However, that is the SQL default and need not be stated explicitly.

The read-committed isolation level, as its name implies, forbids the reading of dirty (uncommitted) data. However, it does allow a transaction running at this isolation level to issue the same query several times and get different answers, as long as the answers reflect data that has been written by transactions that already committed.

Example 6.46: Let us reconsider the seat-choosing program of Example 6.45, but suppose we declare it to run with isolation level read-committed. Then when it searches for a seat at Step (1), it will not see seats as booked if some other transaction is reserving them but not committed.[8] However, if the traveler rejects seats, and one execution of the function queries for available seats many times, it may see a different set of available seats each time it queries, as other transactions successfully book seats or cancel seats in parallel with our transaction. □

Now, let us consider isolation level repeatable-read. The term is something of a misnomer, since the same query issued more than once is not quite guaranteed to get the same answer. Under repeatable-read isolation, if a tuple is retrieved the first time, then we can be sure that the identical tuple will be retrieved again if the query is repeated. However, it is also possible that a second or subsequent execution of the same query will retrieve *phantom* tuples. The latter are tuples that result from insertions into the database while our transaction is executing.

Example 6.47: Let us continue with the seat-choosing problem of Examples 6.45 and 6.46. If we execute this function under isolation level repeatable-read,

[8]What actually happens may seem mysterious, since we have not addressed the algorithms for enforcing the various isolation levels. Possibly, should two transactions both see a seat as available and try to book it, one will be forced by the system to roll back in order to break the deadlock (see the box on "Application- Versus System-Generated Rollbacks" in Section 6.6.3).

then a seat that is available on the first query at Step (1) will remain available at subsequent queries.

However, suppose some new tuples enter the relation `Flights`. For example, the airline may have switched the flight to a larger plane, creating some new tuples that weren't there before. Then under repeatable-read isolation, a subsequent query for available seats may also retrieve the new seats. □

Figure 6.17 summarizes the differences between the four SQL isolation levels.

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not Allowed	Allowed	Allowed
Repeatable Read	Not Allowed	Not Allowed	Allowed
Serializable	Not Allowed	Not Allowed	Not Allowed

Figure 6.17: Properties of SQL isolation levels

6.6.7 Exercises for Section 6.6

Exercise 6.6.1: This and the next exercises involve certain programs that operate on the two relations

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
```

from our running PC exercise. Sketch the following programs, including SQL statements and work done in a conventional language. Do not forget to issue `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK` statements at the proper times and to tell the system your transactions are read-only if they are.

a) Given a speed and amount of RAM (as arguments of the function), look up the PC's with that speed and RAM, printing the model number and price of each.

b) Given a model number, delete the tuple for that model from both `PC` and `Product`.

c) Given a model number, decrease the price of that model PC by $100.

d) Given a maker, model number, processor speed, RAM size, hard-disk size, and price, check that there is no product with that model. If there is such a model, print an error message for the user. If no such model existed in the database, enter the information about that model into the `PC` and `Product` tables.

! **Exercise 6.6.2:** For each of the programs of Exercise 6.6.1, discuss the atomicity problems, if any, that could occur should the system crash in the middle of an execution of the program.

! **Exercise 6.6.3:** Suppose we execute as a transaction T one of the four programs of Exercise 6.6.1, while other transactions that are executions of the same or a different one of the four programs may also be executing at about the same time. What behaviors of transaction T may be observed if all the transactions run with isolation level `READ UNCOMMITTED` that would not be possible if they all ran with isolation level `SERIALIZABLE`? Consider separately the case that T is any of the programs (a) through (d) of Exercise 6.6.1.

!! **Exercise 6.6.4:** Suppose we have a transaction T that is a function which runs "forever," and at each hour checks whether there is a PC that has a speed of 3.5 or more and sells for under $1000. If it finds one, it prints the information and terminates. During this time, other transactions that are executions of one of the four programs described in Exercise 6.6.1 may run. For each of the four isolation levels — serializable, repeatable read, read committed, and read uncommitted — tell what the effect on T of running at this isolation level is.

6.7 Summary of Chapter 6

✦ *SQL*: The language SQL is the principal query language for relational database systems. The most recent full standard is called SQL-99 or SQL3. Commercial systems generally vary from this standard.

✦ *Select-From-Where Queries*: The most common form of SQL query has the form select-from-where. It allows us to take the product of several relations (the `FROM` clause), apply a condition to the tuples of the result (the `WHERE` clause), and produce desired components (the `SELECT` clause).

✦ *Subqueries*: Select-from-where queries can also be used as subqueries within a `WHERE` clause or `FROM` clause of another query. The operators `EXISTS`, `IN`, `ALL`, and `ANY` may be used to express boolean-valued conditions about the relations that are the result of a subquery in a `WHERE` clause.

✦ *Set Operations on Relations*: We can take the union, intersection, or difference of relations by connecting the relations, or connecting queries defining the relations, with the keywords `UNION`, `INTERSECT`, and `EXCEPT`, respectively.

✦ *Join Expressions*: SQL has operators such as `NATURAL JOIN` that may be applied to relations, either as queries by themselves or to define relations in a `FROM` clause.

✦ *Null Values*: SQL provides a special value NULL that appears in components of tuples for which no concrete value is available. The arithmetic and logic of NULL is unusual. Comparison of any value to NULL, even another NULL, gives the truth value UNKNOWN. That truth value, in turn, behaves in boolean-valued expressions as if it were halfway between TRUE and FALSE.

✦ *Outerjoins*: SQL provides an OUTER JOIN operator that joins relations but also includes in the result dangling tuples from one or both relations; the dangling tuples are padded with NULL's in the resulting relation.

✦ *The Bag Model of Relations*: SQL actually regards relations as bags of tuples, not sets of tuples. We can force elimination of duplicate tuples with the keyword DISTINCT, while keyword ALL allows the result to be a bag in certain circumstances where bags are not the default.

✦ *Aggregations*: The values appearing in one column of a relation can be summarized (aggregated) by using one of the keywords SUM, AVG (average value), MIN, MAX, or COUNT. Tuples can be partitioned prior to aggregation with the keywords GROUP BY. Certain groups can be eliminated with a clause introduced by the keyword HAVING.

✦ *Modification Statements*: SQL allows us to change the tuples in a relation. We may INSERT (add new tuples), DELETE (remove tuples), or UPDATE (change some of the existing tuples), by writing SQL statements using one of these three keywords.

✦ *Transactions*: SQL allows the programmer to group SQL statements into transactions, which may be committed or rolled back (aborted). Transactions may be rolled back by the application in order to undo changes, or by the system in order to guarantee atomicity and isolation.

✦ *Isolation Levels*: SQL defines four isolation levels called, from most stringent to least stringent: "serializable" (the transaction must appear to run either completely before or completely after each other transaction), "repeatable-read" (every tuple read in response to a query will reappear if the query is repeated), "read-committed" (only tuples written by transactions that have already committed may be seen by this transaction), and "read-uncommitted" (no constraint on what the transaction may see).

6.8 References for Chapter 6

Many books on SQL programming are available. Some popular ones are [3], [5], and [7]. [6] is an early exposition of the SQL-99 standard.

SQL was first defined in [4]. It was implemented as part of System R [1], one of the first generation of relational database prototypes.