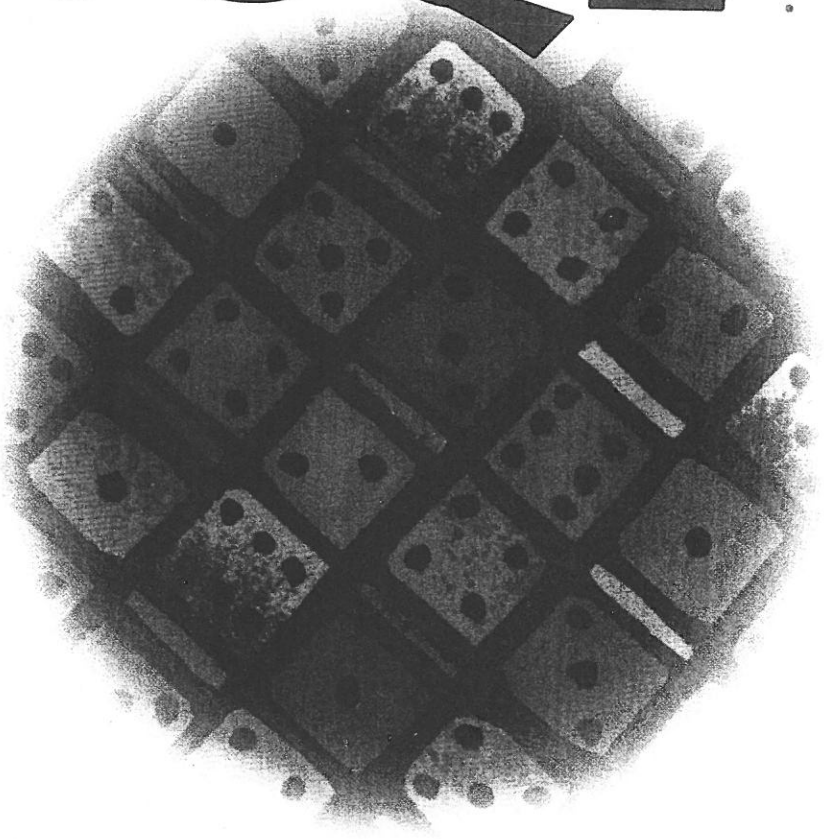
WILEY

TIMELY. PRACTICAL. RELIABLE.

A Visual Introduction to SQL

Second Edition



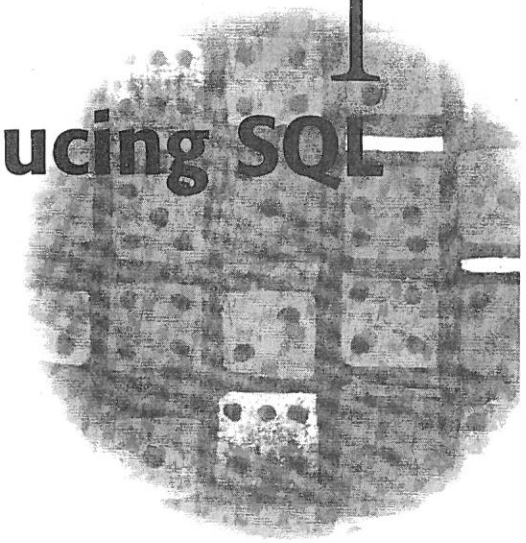
David Chappell
J. Harvey Trimble Jr.

responsible for frequent traveler information systems planning and development. Prior to his work at Marriott, he worked for the Bureau of Labor Statistics where he was responsible for building the information system behind the 1987 revision of the Consumer Price Index.

Mr. Trimble holds a B.S. in accounting from Lehigh University and an M.S. in technology management from The American University.

CHAPTER 1

Introducing SQL



Storing and retrieving information—working with databases—is at the heart of how computers are used today. Databases contain information about people, policies, parts, and a plethora of other things. To make it easier to work with this information, we use software called a *database management system* (DBMS). A typical DBMS allows its users to store, modify, and access data in an organized, efficient way.

Once information has been stored in a DBMS, the next problem is to provide some way to access and work with that information. The most common way to do this is to create a language of some kind that allows users to get at the data in a database. Access to a database is often called a *query*, and so a language that allows this is called a *query language*. While many query languages have been created for various DBMSs, one has become dominant. That language is officially called *Structured Query Language*, but it's more commonly known by its acronym: SQL. (To make things still more confusing, it's pronounced "sequel," as if the acronym contained vowels.) Today, SQL has become the standard database query language, one that's supported by a great many DBMSs.

Kinds of Database Systems

Database management systems can be characterized by the way they model data as seen by their users. In the 1970s, a widely used approach was to organize the data into a hierarchy. Not too surprisingly, DBMSs using this model

were called *hierarchical* systems. A very popular example of a hierarchical system was IBM's IMS (which despite its age is still a key technology in many organizations).

With hierarchical systems, the direct user of the database was typically a programmer, and access to the stored information occurred via special calls from programming languages. The organization of these kinds of systems lends itself to fast access by specially written programs. Even today, hierarchical DBMSs are sometimes used to support fast access to stored data by software. Allowing simple ad hoc queries, however, is not very easy with these kinds of systems. And users, the people who pay for all of this, want some simpler way to work with data.

In part, because of this, another approach appeared that has become the norm for DBMSs today. Called *relational* systems, these newer DBMSs represent data to their users as simply the contents of one or more tables. Among the best-known relational products are IBM's DB2, Oracle Corporation's Oracle DBMS, and Microsoft's SQL Server, although many more exist. SQL was developed as a query language for relational database systems. Although it is occasionally used to access other types of systems, its primary intent is to allow easy access to data stored in relational systems. Virtually all relational systems, including those previously listed, give their users access to stored data through SQL. Together, the advent of relational systems and the wide availability of SQL allow fast answers to an assortment of very specific questions about data stored in a database.

Relational Concepts

The relational model was first put forth by E. F. Codd, a researcher in IBM's San Jose laboratories, in a paper published in 1970. Since then, the relational approach to database management has become dominant in the industry. A major reason for this dominance is the simple but powerful view of data that relational systems offer their users. Software developers can easily understand how data is organized, and so developing applications that use SQL to access that data is straightforward. Even end users can potentially create their own interactive SQL queries, although the language can get a bit complex at times. While the language has its critics, the notion of a relational database today has become almost inseparable from SQL.

Tables, Rows, and Columns

Perhaps the simplest way to think of a relational DBMS is as a system in which users see all data as the contents of one or more *tables* and nothing more. All

data in the database is stored in one of these tables. Each table is a simple two-dimensional structure made up of some number of *rows* (also called *records*) and *columns* (also called *fields*). Each column in a table is assigned a unique name and contains a particular type of data, such as characters or numbers. Each record contains a value for each of the table's columns.

Stored in the table as values for the various fields in the records is the actual data. Note that the order of the records in a table is not significant. It is not meaningful to ask whether a particular record occurs before or after another. The notion of a table, with its rows and columns, is illustrated in Figure 1.1.

A very simple database might contain only one table, but most databases will contain several. For instance, imagine a sales manager who would like to determine how each of her salespeople is performing. The database she accesses might hold in one table the amount of each product sold by each member of her sales force while another table contains personnel information about the salespeople, such as their length of service with the company. Still other tables in this database will contain other related information. Some queries against this database, such as determining which salesperson sold the most of a particular product, can be answered by examining a single table.

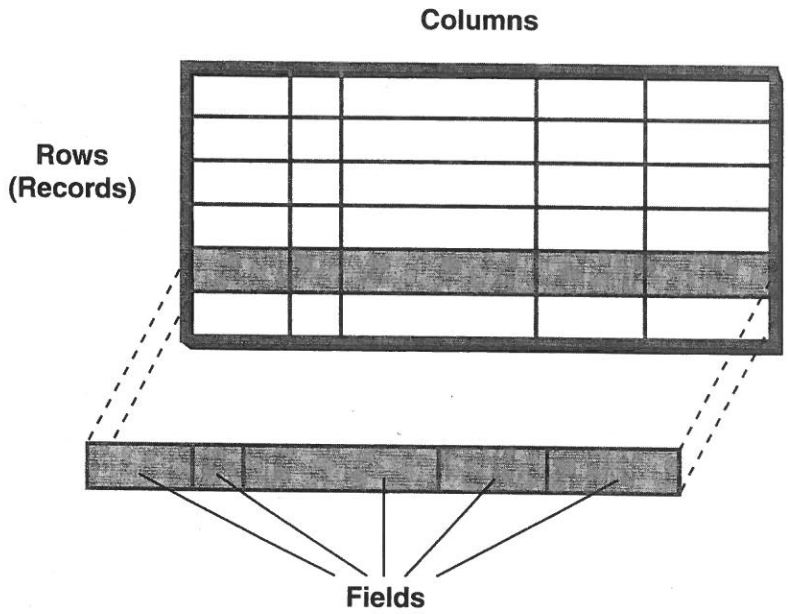


Figure 1.1 Rows, columns, and fields in a table.

Others, such as requesting the total sales last month by each sales person with the results ranked by length of service, might require accessing more than one table. Neither of these presents a problem, since a single SQL query can retrieve information from one or many tables within a database.

Keys

If the records in a table are not ordered, how are we to locate specific ones? And how can we distinguish one record from another? The answer lies in specifying one or more columns in each table that define the *key* for that table's records. The information in the key field or fields must be unique for each record. For example, in the sales database just described, the table containing information about each salesperson might designate the column containing the person's name as its key. Since it's possible that two members of the sales force have the same name, however, it would probably be better to store some other value in each salesperson record that's guaranteed to be unique for each member of the sales force. One obvious possibility for this value, at least in the United States, is the salesperson's Social Security number.

For some tables, none of the information to be stored might provide an appropriate key. In these cases, it is necessary to define a column containing some kind of unique identification (typically a number) to serve as the key. For other tables, the contents of two or more columns taken together can provide the values necessary to uniquely identify each record. In tables like these, a record's key consists of its values in both of those columns.

Keys are not always required for accessing information. Depending on the question asked, it might or might not be necessary to refer to each record's key value. In spite of this, the ability to uniquely identify each record in a table is an integral part of the relational model for handling data, and every table must always define a key for its records.

Correctly selecting keys for the tables in a database is not an easy task. In fact, choosing keys is only one aspect of the much larger problem of designing a relational database. The designer, who might also be the *database administrator* (DBA) responsible for this system, must first determine how the data should be organized into tables. The best organization will depend on exactly how that data is to be accessed. The DBA (or whoever the designer is) must determine the number of columns in each table, what the names and allowable contents of those columns should be, and many other specifics about the database. Luckily, accessing the data through SQL doesn't require any specialized knowledge about how a relational database should be defined. All a software developer or user needs to understand is how the tables in the database look to them.

teacher#	teacher_name	phone	salary
303	Dr. Horn	257-3049	27540.00
290	Dr. Lowe	257-2399	31450.00
430	Dr. Engle	256-4621	38200.00
180	Dr. Cooke	257-8088	29560.00
560	Dr. Olsen	257-8086	31778.00
784	Dr. Scango	257-3046	32098.00

Figure 1.2 The `TEACHERS` table.

An Example Table

Figure 1.2 shows the contents of a table called `TEACHERS`. (Throughout this book, we show the names of tables in all upper case. This convention is adopted strictly for clarity; SQL in general imposes no such requirement.) This table contains six rows, each a record describing a particular teacher. The information stored about each is described by the names assigned to the table's four columns: `teacher#`, `teacher_name`, `phone`, and `salary`. The `teacher#` column is designated as the key for this table.

Every record in the table has four fields, each containing a value for one of these four columns. The first record in the table, for instance, contains a value of 303 for the column called `teacher#`, 'Dr. Horn' for the column `teacher_name`, '257-3049' for the column `phone`, and 27540.00 for the last column, `salary`. Each of the other records in this table contains similar values.

The objective of SQL is to allow you to work with the information stored in a relational database's tables. Using SQL, you can create new tables, destroy existing ones, and modify a table's records. Most importantly, though, you can ask questions about the information stored in the tables, questions such as "How much money does Dr. Olsen make?" and "What is Dr. Lowe's phone number?" It is these sorts of questions that make up the bulk of SQL queries in most database applications.

The Visual Approach

While SQL is a very powerful tool, it can also be rather complex to use. Among the major reasons for this are the formal nature of the language (it is, after all,

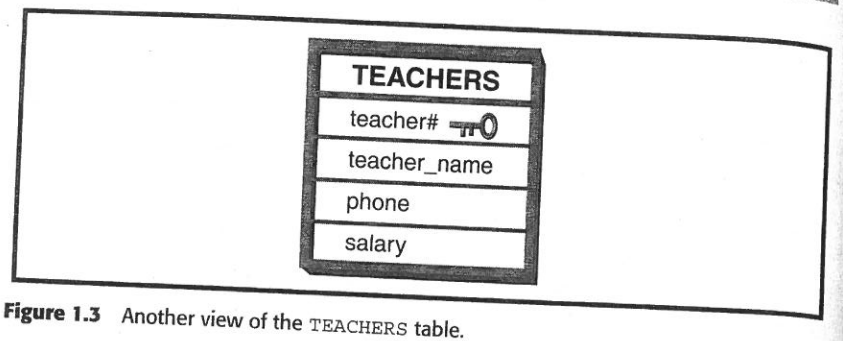


Figure 1.3 Another view of the TEACHERS table.

still a computer language) and the inherent complexity of the queries it can express. To tame this complexity, this book takes a visual approach. We believe that by first thinking of a query *visually*, in terms of query diagrams, you can more easily develop the formal SQL statements required to actually execute that query against a real relational database.

In the visual approach, each table is represented as a box with each of the column names listed. Figure 1.3 shows how the TEACHERS table looks using this representation. (Note the little key next to teacher#, indicating that this column contains the key values for TEACHERS.)

Queries about the information in a table are shown by graphically indicating the information to be retrieved. For example, Figure 1.4 is a diagram of the query, "What are the names and phone numbers of all teachers earning more than $30,000 a year?"

That same query expressed in SQL is:

```
SELECT teacher_name, phone
  FROM TEACHERS
  WHERE salary > 30000;
```

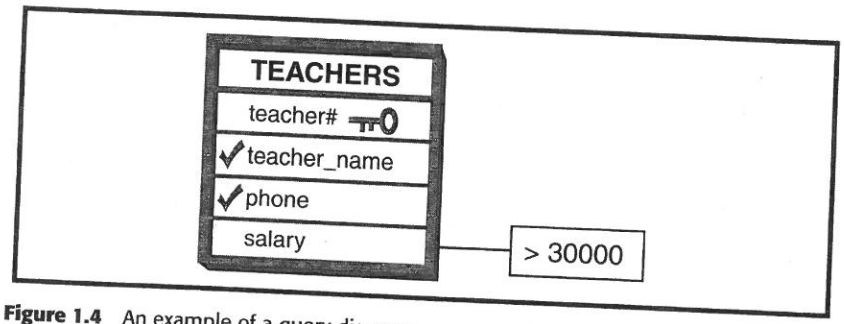


Figure 1.4 An example of a query diagram.

For many queries, information must be retrieved from two or more tables at once. Correctly formulating these kinds of requests directly in SQL can be especially difficult. In this book, we attempt to make both these and other queries easier and more intuitive by first showing a query diagram, then presenting the actual SQL.

Starting with simple queries, we will move step by step to increasingly powerful operations. By first visualizing each query, then showing the SQL and results, we believe that the query's underlying structure and ultimately the structure of SQL itself will be made clear.

but their usefulness becomes increasingly apparent as the queries get more complex. By presenting the diagrams *before* the actual SQL statements, we hope to make intuitively obvious how the query should be formulated. Following each diagram comes the SQL query itself, and finally, we present the results of executing that query against the example database.

This organization was not chosen arbitrarily. By rigorously following this pattern, we hope to instill a methodology for approaching query development, one that is most likely to lead to the correct expression of the query in SQL.

CHAPTER 3

Retrieving Data from a Table: The SELECT Statement

By far the most used of SQL's verbs is SELECT. Since the most common operation performed on a database is to examine its data, it should not be surprising that the SQL statement that performs this task is the workhorse of the language. In any real database, of course, other SQL statements would have to be used first to create the database's tables and fill them with records. For most SQL users, however, this task will already have been performed, and so we begin with SELECT. (If you wish first to create tables or add records to existing tables, skip ahead to Chapter 7, "Creating and Destroying Tables," and Chapter 8, "Adding, Modifying, and Deleting Records.")

Selecting Specific Columns in a Table

To see the values of certain columns for all of a table's records, you must give both the names of those columns and the name of the table. The general form is

```
SELECT <column names>
    FROM <table>;
```

where <column names> is replaced by the names of the desired columns, separated by commas, and <table> is replaced by the name of a table that contains those columns.

NOTE

Throughout this book, words flanked by < and > represent placeholders for a type of thing and are replaced by specific instances of those things in actual SQL queries.

In general, if you specify a column that is not defined for that table, you will get some kind of error message depending on exactly what system you are using. Otherwise, the results of your query, a list of the desired values, will appear. To show these results visually, the query diagram contains a representation of the table being queried with a check mark next to each column selected by the query.

Example: Selecting specific columns from a table

Problem

List the names of all courses, their departments, and the number of credits for each.

Query Diagram

SQL

```
SELECT course_name, department, num_credits
    FROM COURSES;
```

Results

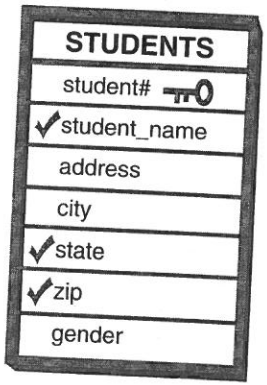
```
course_name                department                num_credits
-------------------------- ------------------------- -----------
Western Civilization       History                   3
Calculus IV                Math                      4
English Composition        English                   3
Compiler Writing           Computer Science          3
Art History                History                   3
```

Example: Selecting specific columns from a table

Problem

List the names, hometowns, and home states of all students.

Query Diagram



SQL

```
SELECT student_name, city, state
    FROM STUDENTS;
```

Results

student_name	city	state
Susan Powell	Haverford	PA
Bob Dawson	Newport	RI
Howard Mansfield	Vienna	VA
Susan Pugh	Hartford	CT
Joe Adams	Newark	DE
Janet Ladd	Pennsburg	PA
Bill Jones	Newport Beach	CA
Carol Dean	Boston	MA
Allen Thomas	Chicago	IL
Val Shipp	Chicago	IL
John Anderson	New York	NY
Janet Thomas	Erie	PA

Selecting All Columns in a Table

It is often useful to see the value of every field for every record in a table. One way to do this is by listing the names of every column in that table, similar to the examples we have just seen. Because this is such a frequent operation, however, SQL provides a shorthand way to list all values in a table.

Instead of actually listing all column names, you can type an asterisk instead. The general form is

```
SELECT *
    FROM <table>;
```

where `<table>` is replaced by the name of a table. Unsurprisingly, the query diagram for this shows the table with a check by every column name.

Example: Selecting all columns in a table

Problem

List all values in the COURSES table.

Query Diagram

SQL

```
SELECT *
    FROM COURSES;
```

Results

course#	course_name	department	num_credits
450	Western Civilization	History	3
730	Calculus IV	Math	4
290	English Composition	English	3
480	Compiler Writing	Computer Science	3
550	Art History	History	3

Example: Selecting all columns in a table

Problem

List all values in the TEACHERS table.

Query Diagram

SQL

```
SELECT *
    FROM TEACHERS;
```

Results

teacher#	teacher_name	phone	salary
303	Dr. Horn	257-3049	27540.00
290	Dr. Lowe	257-2390	31450.00
430	Dr. Engle	256-4621	38200.00
180	Dr. Cooke	257-8088	29560.00
560	Dr. Olsen	257-8086	31778.00
784	Dr. Scango	257-3046	32098.00
213	Dr. Wright	257-3393	35000.00

Selecting Only Some of a Table's Records: The WHERE Clause

So far, every SELECT statement has returned at least one value for every record in the table. What if we wish to see values only for records that meet some specific criteria? To do this, we must use the SELECT statement's WHERE clause. The WHERE clause lets us specify a *predicate*, something that is either true or false about each record in the table. Only those records for which the predicate is true will be listed in the results.

The general form of a SELECT using WHERE is

```
SELECT <column names>
    FROM <table>
    WHERE <predicate>;
```

As before, <column names> and <table> are replaced by appropriate column and table names. <predicate> can be replaced by a number of different things, depending on exactly what restrictions you wish to place on the results. In the remainder of this chapter, we will examine the possible restrictions allowed by predicates.

The query diagrams for SELECTs with a WHERE clause indicate the WHERE's restriction in a box attached to the restricted column. For each type of predicate, the box contains an appropriate description of the restriction applied to that column.

Comparisons in a WHERE Clause

Probably the most common predicates are those that compare values. You might wish, for instance, to see only the records in the COURSES table for three credit courses or only those in the TEACHERS table for teachers with salaries greater than $30,000 a year. (You will notice that this university doesn't believe in overpaying its faculty). For these types of queries, the general form is

```
SELECT <column names>
    FROM <table>
    WHERE <column name> <operator> <value>;
```

As before, <column names> and <table> represent the names of the desired columns and the name of the table from which you should draw them. The WHERE clause is more complex, however. The first item, <column name>, must name a particular column in the table. This column might or might not be among those listed in <column names>.

Table 3.1 Comparison Operators

OPERATOR	FUNCTION
=	true if the value contained in <column name> equals the value given in the WHERE clause
<>	true if the value contained in <column name> is not equal to that given in the WHERE clause (some systems use other symbols, such as !=, in place of <>)
<	true if the value contained in <column name> is less than the value given in the WHERE clause
>	true if the value contained in <column name> is greater than the value given in the WHERE clause
<=	true if the value contained in <column name> is less than or equal to that given in the WHERE clause
>=	true if the value contained in <column name> is greater than or equal to that given in the WHERE clause

After the <column name> but before the <value> comes an <operator>. The choices for this operator, known as a *comparison* operator, are shown in Table 3.1.

The predicate ends with a <value>. The exact form of this value varies depending on the type of the named column. For columns of numeric types, a numeric value is simply placed after the operator. If the column is CHARACTER (CHAR) or CHARACTER VARYING (VARCHAR), however, the characters comprising the value must be enclosed in single quotes.

Although SQL is usually not sensitive to the difference between upper and lower-case letters, case does make a difference for comparisons using quoted character values. For example, the character strings 'Hello' and 'HELLO' are not considered equal to one another. Also, some strings are longer than others. According to the SQL standard, comparing two strings of unequal length conceptually adds blanks to the end of the shorter string and then performs the comparison. And, like numeric values, you can ask whether one character string is less than or greater than another. The comparison is performed based on the collating sequence used by your system. In general, this collating sequence for an English-language installation of SQL will result in a normal alphabetic comparison, with 'A' less than 'B', 'B' less than 'C', and so on. (Whether digits are less than letters varies, though, depending on the character set specified as the default for the implementation.)

In some cases, the <value> might also be a <column name>. If the <table> identifies a column in <column name>, the values in the two columns are

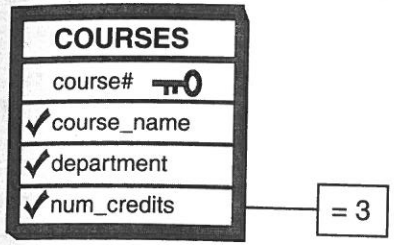
compared for each record; only those records in which the two values satisfy the condition (e.g., =) are returned. Alternatively, the `<column name>` might identify a column in a *different* table. In this case, records from both tables can be examined and retrieved. Retrieving data from more than one table at a time is called a *join* and is discussed in Chapter 5, "Retrieving Data from Several Tables: Joins."

Example: Comparisons using a WHERE clause

Problem

List the course name, department, and number of credits for all three-credit courses.

Query Diagram



SQL

```
SELECT course_name, department, num_credits
    FROM COURSES
    WHERE num_credits = 3;
```

Results

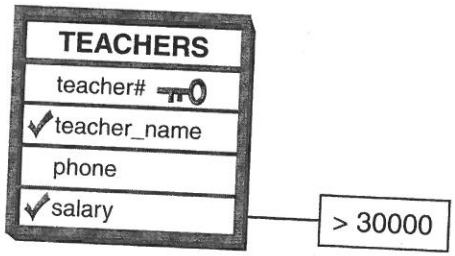
course_name	department	num_credits
Western Civilization	History	3
English Composition	English	3
Compiler Writing	Computer Science	3
Art History	History	3

Example: Comparisons using a WHERE clause

Problem

List the names and salaries of teachers earning more than $30,000.

Query Diagram



SQL

```
SELECT teacher_name, salary
    FROM TEACHERS
    WHERE salary > 30000;
```

Results

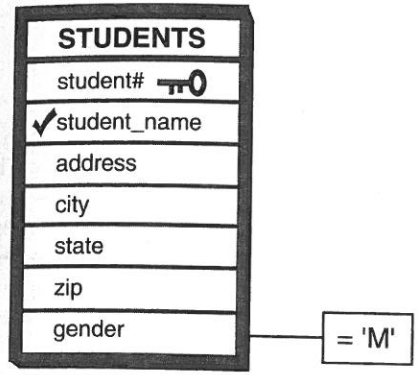
```
teacher_name          salary

Dr. Lowe              31450.00
Dr. Engle             38200.00
Dr. Olsen             31778.00
Dr. Scango            32098.00
Dr. Wright            35000.00
```


Example: Comparisons using a WHERE clause

Problem

Who are all the male students?

Query Diagram



SQL

```
SELECT student_name
    FROM STUDENTS
    WHERE gender = 'M';
```

Results

```
student_name

Bob Dawson
Howard Mansfield
Joe Adams
Bill Jones
Allen Thomas
John Anderson
```

Combining Predicates in a WHERE Clause

It is often useful to combine two or more predicates in a single WHERE clause. For example, you might want to identify all students who both live in Chicago and are women. Performing this action requires using SQL's AND and OR operators. AND and OR, together with a third operator called NOT, are known as boolean operators.

Using AND

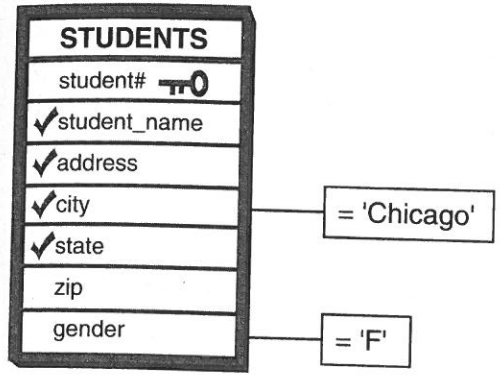
SQL uses the word AND in much the same way as English does. Two predicates can be combined with AND, and the entire predicate is true only if both of its parts are also true. If desired, either or both of the predicates can be enclosed in parentheses. We will soon see examples where parentheses are required, but for now they are optional. Visually, AND is indicated by showing both conditions. If more than one condition is shown in a query diagram, you can assume that they are connected together in the actual SQL query with AND.

Example: Using AND in a WHERE clause

Problem

List the names and addresses of all female students from Chicago.

Query Diagram



SQL

```
SELECT student_name, address, city, state
    FROM STUDENTS
    WHERE city = 'Chicago' AND gender = 'F';
```

Results

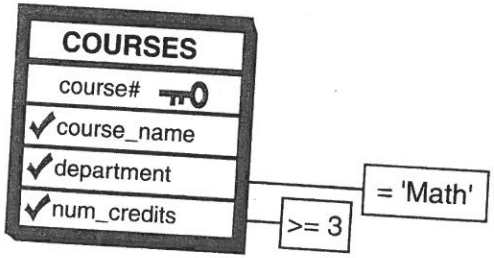
student_name	address	city	state
Val Shipp	238 Westport Road	Chicago	IL

Example: Using AND in a WHERE clause

Problem

Which math courses have three or more credits?

Query Diagram



SQL

```
SELECT course_name, department, num_credits
    FROM COURSES
    WHERE department = 'Math' AND num_credits >= 3;
```

Results

course_name	Department	num_credits
Calculus IV	Math	4

Using OR

OR is used in a similar way to AND. Once again, two predicates can be combined, and the value of both is used in determining which records are returned from a table. With AND, both predicates had to be true for a particular record to be selected. With OR, on the other hand, a record is selected and appears in the results of the query if either of the predicates is true or if both are true. Because it is used less often than AND, OR is explicitly indicated in query diagrams.

AND and OR can both be used in the same WHERE clause. In fact, arbitrarily complex expressions can be created by using various combinations of AND and OR. If both AND and OR appear in the same WHERE clause, all of the ANDs are evaluated first, followed by all of the ORs. There is one exception to this rule: Anything enclosed in parentheses will be executed first. In some cases, then, you might be required to use parentheses to correctly express what you mean.

For example, to see the names of all male students who live in either Connecticut or New York, you could type

```
SELECT student_name
    FROM STUDENTS
    WHERE (state = 'CT' OR state = 'NY') AND
          gender = 'M';
```

If the parentheses were omitted, the statement would become

```
SELECT student_name
    FROM STUDENTS
    WHERE state = 'CT' OR state = 'NY' AND
          gender = 'M';
```

It now has a quite different meaning. Because SQL first evaluates all ANDs, it is as if the query were written

```
SELECT student_name
    FROM STUDENTS
    WHERE state = 'CT' OR
          (state = 'NY' AND gender = 'M');
```

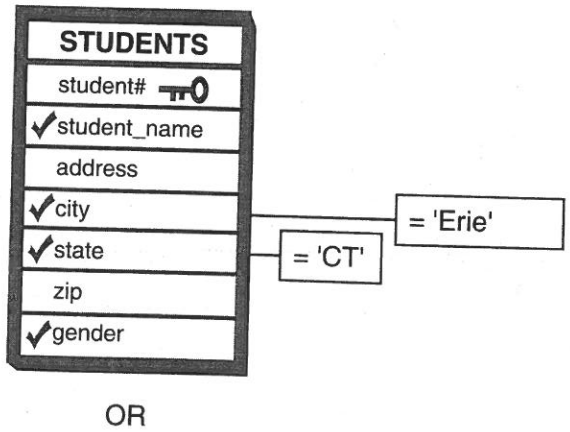
This query will select the records of all male students from New York as well as those of all students, both male and female, from Connecticut. The moral is this: Use parentheses whenever you combine AND and OR in a WHERE clause. They never hurt, and they sometimes can prevent you from inadvertently making the wrong request.

Example: Using OR in a WHERE clause

Problem

List the name, gender, city, and state for all students who are either from Connecticut or from a city called Erie.

Query Diagram



SQL

```
SELECT student_name, gender, city, state
    FROM STUDENTS
    WHERE state = 'CT' OR
          city = 'Erie';
```

Results

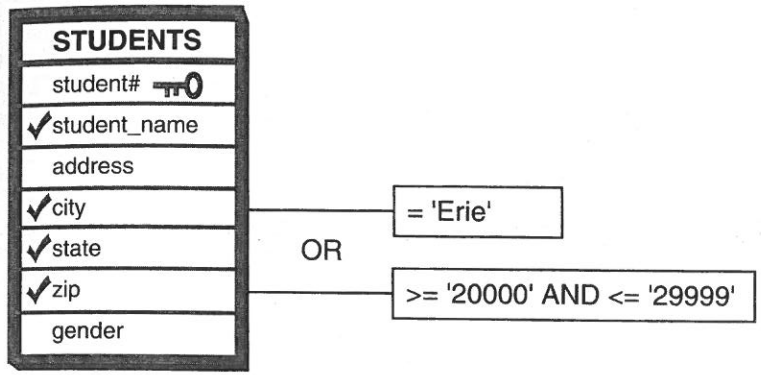
student_name	gender	city	state
Susan Pugh	F	Hartford	CT
Janet Thomas	F	Erie	PA

Example: Using both AND and OR in a WHERE clause

Problem

List the names, cities, states, and zip codes of all students whose zip codes are between 20000 and 29999 or who live in a city called Erie.

Query Diagram



SQL

```
SELECT student_name, city, state, zip
    FROM STUDENTS
    WHERE (zip >= '20000' AND zip <= '29999') OR
          city = 'Erie';
```

Results

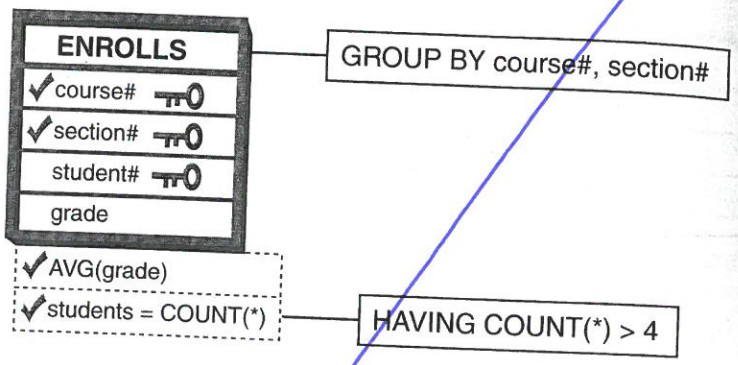
student_name	city	state	zip
Howard Mansfield	Vienna	VA	22180
Janet Thomas	Erie	PA	16510

Example: Specifying a property for grouped data with HAVING

Problem

List the average grade and number of students for each section with more than four students.

Query Diagram



SQL

```
SELECT course#, section#, AVG(grade), COUNT(*) AS students
    FROM ENROLLS
    GROUP BY course#, section#
    HAVING COUNT(*) > 4;
```

Results

course#	section#	AVG(grade)	students
730	1	2.67	6

CHAPTER 5

Retrieving Data from Several Tables: Joins

Every query we have seen so far retrieves data from only a single table. If all of a database's information were contained in one large table, these types of queries would be all that was required. Practically, though, storing all the information in just one table would require maintaining several duplicate copies of the same values. In real systems, therefore, all but the very simplest databases divide their information among several different tables.

Both in real databases and in our simple example database, then, many interesting questions cannot be answered by retrieving data from only a single table. Instead, we must form queries that simultaneously access two or more tables. Any query that extracts data from more than one table must perform a *join*. As its name suggests, a join means that some or all of the specified tables' contents are joined together in the results of the query.

Qualified Names

Before tackling joins, we need to first describe the notion of *qualified names*. As we've already described, every SELECT statement must indicate the names of the columns that should be returned. So far, all of those columns have been from the same table. With joins, however, we will be selecting columns from two or more tables simultaneously. Although all of the columns in a single

table must have unique names, columns in different tables can have the same name. To identify a column in the database uniquely, then, we must use its qualified name.

A qualified name consists of the name of a table followed by a period and the name of a column in that table. For example, the qualified name for the teacher# column in the SECTIONS table is SECTIONS.teacher#, while the qualified name for the teacher# column in the TEACHERS table is TEACHERS.teacher#. Because every column in a table must have a different name, and because SQL requires every table in a database to have a different name, a qualified name is guaranteed to identify exactly one column in exactly one table.

To see how qualified names are used in a query, suppose that we wished to see each value for teacher_name from TEACHERS along with each possible value for teacher# and course# from SECTIONS. Requesting teacher_name from TEACHERS along with course# from SECTIONS is easy: Both column names are unique in the database. But selecting teacher# from SECTIONS requires using the qualified name for teacher#, because a column with that name appears in both tables. A formulation of this query might be

```
SELECT teacher_name, SECTIONS.teacher#, course#
    FROM TEACHERS, SECTIONS;
```

(What are the results of this query? Probably not what you would expect—see the next section.)

As we have already seen, qualified names are not required when there is no possibility of ambiguity. In the previous query, for instance, just the column name was required for course# because TEACHERS and SECTIONS have only one column with this name between them. Despite the fact that a column called course# also appears in the COURSES table, the list of tables in the previous query's FROM clause limits its scope to TEACHERS and SECTIONS.

We have been taking advantage of this automatic limitation all along. For instance, in the query

```
SELECT student#
    FROM STUDENTS;
```

student# must refer to the column of that name in the STUDENTS table. Similarly, in the query

```
SELECT teacher#
    FROM TEACHERS;
```


there is still no ambiguity because the FROM clause specifies from which table the values for teacher# are to be drawn. Even simple queries can use qualified names if desired, such as

```
SELECT STUDENTS.student_name
    FROM STUDENTS;
```

although it's hard to see why this option would be useful. In the first example given previously, however, ambiguity was possible, so a qualified name was required.

What Is a Join?

If you issue the query

```
SELECT teacher_name, SECTIONS.teacher#, course#
    FROM TEACHERS, SECTIONS;
```

what will come back in return? Obviously, the result will have three columns: teacher_name, SECTIONS.teacher#, and course#. But what values will be contained in those three columns?

To carry out this query, SQL will list values from each record of both tables. In fact, it will list all possible combinations of the selected columns from all records in the two tables. In other words, the results will begin with a line containing the values of teacher# and course# from the first record in SECTIONS matched with the value of teacher_name from the first record in TEACHERS. Next will come a line matching the next teacher_name with the selected values from the first record in the SECTIONS table, and so on, until each of the records in the TEACHERS table has been matched with the first SECTIONS record. Next, the results will contain a series of lines that match the selected values from the second record in SECTIONS with the teacher_name value from every record in the TEACHERS table, and so on. The result of this seemingly simple query is a long list of combinations, each consisting of a value from a record in TEACHERS paired with two values from a record in SECTIONS.

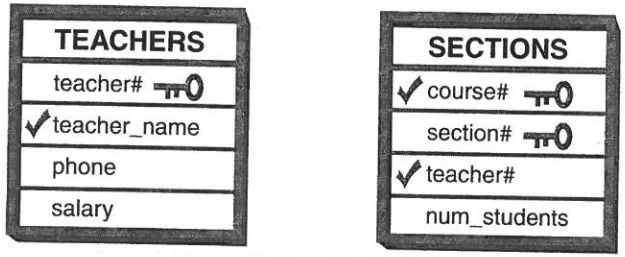
In relational database theory, this long list of combinations is called a *cross join*, or even more technically a *Cartesian product*. Cross joins are used fairly rarely, but they can be useful in some situations. (Most often, however, they're produced by accident.)

Example: A simple join

Problem

List all teacher names along with all the values for teacher number and course number contained in the SECTIONS table.

Query Diagram



SQL

```
SELECT teacher_name, SECTIONS.teacher#, course#
    FROM TEACHERS, SECTIONS;
```

Results

teacher_name	SECTIONS.teacher#	course#
Dr. Horn	303	450
Dr. Lowe	303	450
Dr. Engle	303	450
Dr. Cooke	303	450
Dr. Olsen	303	450
Dr. Scango	303	450
Dr. Wright	303	450
Dr. Horn	290	730
Dr. Lowe	290	730
Dr. Engle	290	730
Dr. Cooke	290	730
Dr. Olsen	290	730
Dr. Scango	290	730
Dr. Wright	290	730
Dr. Horn	430	290
Dr. Lowe	430	290
Dr. Engle	430	290
Dr. Cooke	430	290
Dr. Olsen	430	290
Dr. Scango	430	290
Dr. Wright	430	290
Dr. Horn	180	480
Dr. Lowe	180	480
Dr. Engle	180	480
Dr. Cooke	180	480
Dr. Olsen	180	480
Dr. Scango	180	480
Dr. Wright	180	480
Dr. Horn	560	450
Dr. Lowe	560	450
Dr. Engle	560	450
Dr. Cooke	560	450
Dr. Olsen	560	450
Dr. Scango	560	450
Dr. Wright	560	450
Dr. Horn	784	480
Dr. Lowe	784	480
Dr. Engle	784	480
Dr. Cooke	784	480
Dr. Olsen	784	480
Dr. Scango	784	480
Dr. Wright	784	550
Dr. Horn	0	550
Dr. Lowe	0	550
Dr. Engle	0	550
Dr. Cooke	0	550
Dr. Olsen	0	550
Dr. Scango	0	550
Dr. Wright	0	550

Restricting the Results of a Join

What happens if, in a query accessing two tables with columns of the same name, we specify the same column name twice, once for each table? Doing this requires that we give the qualified name for each, such as

```
SELECT teacher_name, TEACHERS.teacher#,
       SECTIONS.teacher#, course#
    FROM TEACHERS, SECTIONS;
```

The results are like those previously specified—a list of all possible combinations of the selected columns from the two tables—except that the column called `teacher#` appears twice. The results of this query are as follows:

teacher_name	TEACHERS.teacher#	SECTIONS.teacher#	course#
Dr. Horn	303	303	450
Dr. Lowe	290	303	450
Dr. Engle	430	303	450
Dr. Cooke	180	303	450
Dr. Olsen	560	303	450
Dr. Scango	784	303	450
Dr. Wright	213	303	450
Dr. Horn	303	290	730
Dr. Lowe	290	290	730
Dr. Engle	430	290	730
Dr. Cooke	180	290	730
Dr. Olsen	560	290	730
Dr. Scango	784	290	730
Dr. Wright	213	290	730
Dr. Horn	303	430	290
Dr. Lowe	290	430	290
Dr. Engle	430	430	290
Dr. Cooke	180	430	290
Dr. Olsen	560	430	290
Dr. Scango	784	430	290
Dr. Wright	213	430	290
Dr. Horn	303	180	480
Dr. Lowe	290	180	480
Dr. Engle	430	180	480
Dr. Cooke	180	180	480
Dr. Olsen	560	180	480
Dr. Scango	784	180	480
Dr. Wright	213	180	480
Dr. Horn	303	560	450
Dr. Lowe	290	560	450
Dr. Engle	430	560	450
Dr. Cooke	180	560	450
Dr. Olsen	560	560	450
Dr. Scango	784	560	450
Dr. Wright	213	560	450
Dr. Horn	303	784	480
Dr. Lowe	290	784	480
Dr. Engle	430	784	480
Dr. Cooke	180	784	480
Dr. Olsen	560	784	480
Dr. Scango	784	784	480
Dr. Wright	213	784	480
Dr. Horn	303	0	550
Dr. Lowe	290	0	550
Dr. Engle	430	0	550
Dr. Cooke	180	0	550
Dr. Olsen	560	0	550
Dr. Scango	784	0	550
Dr. Wright	213	0	550

As always, each column is labeled with the appropriate name, so for the two `teacher#` columns, both column names are qualified names. These two columns contain all possible combinations of values for `teacher#` from the two tables.

Usually, general results such as these are not very useful. Too much information is given, and it is difficult to pick out what is interesting. To reduce the size of the results and thereby zero in on the answer to some particular question, we can add a `WHERE` clause to the previous query. We might wish to know, for example, which teachers are teaching which courses. Further, we might wish to identify those teachers by name, not just by their teacher numbers. This information is available in the results from the previous query, but it's not in a very concise or usable form. By adding a `WHERE` clause, we can retrieve only those results of the join in which we are interested.

To see which teachers teach which courses, look through the results just given. Note that whenever the value of `teacher#` from the `TEACHERS` table is equal to the value of `teacher#` from the `SECTIONS` table, the course whose `course#` appears in that record is one that is taught by that teacher. Because of the way this database is defined, the two `teacher#` columns must have this relationship. Columns like these, sometimes called *join columns*, exist in each of the tables in our example database. Similar columns will also exist in most or all of the tables in real relational databases. The information stored in these join columns gives us a very powerful tool for accessing the stored data.

Back to the original question: Which teachers teach which classes? To select just those records from the mass of information given previously, one could type

```
SELECT teacher_name, SECTIONS.teacher#, course#
    FROM TEACHERS, SECTIONS
    WHERE TEACHERS.teacher# = SECTIONS.teacher#;
```

The results are only those records that meet the condition specified in the `WHERE` clause: Those where the two `teacher#` values are equal. We can now

see easily which courses are taught by each teacher, and the teachers are identified by name.

```
teacher_name        SECTIONS.teacher#       course#

Dr. Horn            303                     450
Dr. Lowe            290                     730
Dr. Engle           430                     290
Dr. Cooke           180                     480
Dr. Olsen           560                     450
Dr. Scango          784                     480
```

Even though they are used in the WHERE clause, the values for teacher# need not be selected by the query. If we were interested in seeing only teachers' names and the numbers of the courses they teach, we would have used

```
SELECT teacher_name, course#
    FROM TEACHERS, SECTIONS
    WHERE TEACHERS.teacher# = SECTIONS.teacher#;
```

This time, the results include only the teacher_name and course# values for those records that meet the WHERE clause's condition; that is, a list of teachers and the courses taught by each.

Because the SECTIONS table contains records only for currently offered sections, we could see only the names of those teachers who are currently teaching some section with

```
SELECT teacher_name
    FROM TEACHERS, SECTIONS
    WHERE TEACHERS.teacher# = SECTIONS.teacher#;
```

The important thing about this example is its FROM clause: Despite the fact that no values from the SECTIONS table are selected, that table must still appear in the FROM clause because a value from SECTIONS *is* referenced in the WHERE clause.

Query Diagrams for Joins

The visual approach to SQL really comes into its own when depicting joins. The columns from the tables being joined, such as TEACHERS.teacher# and SECTIONS.teacher#, are indicated by a line connecting them. This line is marked with the operator used in the WHERE clause (typically an equals sign). For more complex joins involving more than two tables, several such connecting lines will exist between tables.

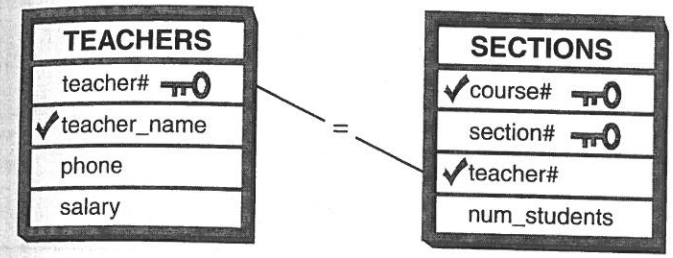
Joins can get complicated. By first thinking of (or actually drawing) the query visually, this complexity becomes manageable.

Example: A join with a simple WHERE clause

Problem

Which teachers teach which courses? (List the teachers by name.)

Query Diagram



SQL

```
SELECT teacher_name, SECTIONS.teacher#, course#
    FROM TEACHERS, SECTIONS
    WHERE TEACHERS.teacher# = SECTIONS.teacher#;
```

Results

```
teacher_name        SECTIONS.teacher#       course#

Dr. Horn            303                     450
Dr. Lowe            290                     730
Dr. Engle           430                     290
Dr. Cooke           180                     480
Dr. Olsen           560                     450
Dr. Scango          784                     480
```

NOTE

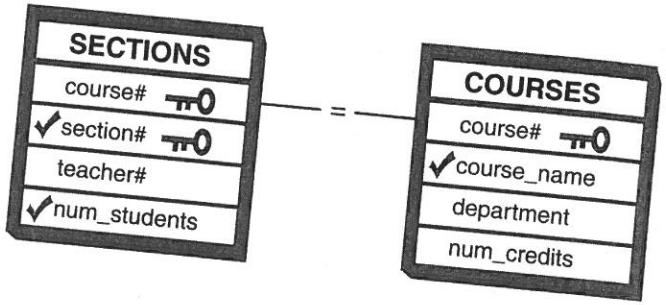
If we were willing to settle for identifying teachers solely by their teacher#, no join would be required. The SECTIONS table alone contains enough information to answer that question. By requesting the *names* of teachers, however, we also request a join, because teacher names and teacher numbers are not stored in the same table.

Example: A join with a simple WHERE clause

Problem

What is the enrollment in each section of each course?

Query Diagram



SQL

```
SELECT section#, course_name, num_students
    FROM SECTIONS, COURSES
    WHERE SECTIONS.course# = COURSES.course#;
```

Results

section#	course_name	Num_students
1	Western Civilization	2
2	Western Civilization	2
1	Calculus IV	6
1	English Composition	3
2	Compiler Writing	3
1	Compiler Writing	2
	Art History	0

Joining Tables Using the Join Operator

In the initial versions of the SQL language, joins could only be created using the WHERE clause, and this syntax is still supported in SQL-92. This newer version, however, also introduces a new explicit JOIN operator. To get a list of which teachers teach which courses using the JOIN operator, the query would be

```
SELECT teacher_name, SECTIONS.teacher#, course#
    FROM TEACHERS INNER JOIN SECTIONS
    ON TEACHERS.teacher# = SECTIONS.teacher#;
```

Compare this query to the syntax using the WHERE clause:

```
SELECT teacher_name, SECTIONS.teacher#, course#
    FROM TEACHERS, SECTIONS
    WHERE TEACHERS.teacher# = SECTIONS.teacher#;
```

Both queries return the same results:

teacher_name	SECTIONS.teacher#	course#
Dr. Horn	303	450
Dr. Lowe	290	730
Dr. Engle	430	290
Dr. Cooke	180	480
Dr. Olsen	560	450
Dr. Scango	784	480

The two tables in the example are combined with the operator INNER JOIN, and the WHERE clause is replaced by an ON clause. An *inner join* is the most common type of join. It returns only those records from each table that match the criteria specified in the ON clause. If you compare the previous results to the complete list of teachers, you'll see that Dr. Wright is not listed because she doesn't currently teach any courses.

We've seen another type of join, the cross join, which returns all possible combinations of the records in both tables. The simplest form of cross join, shown earlier with the syntax

```
SELECT teacher_name, TEACHERS.teacher#, SECTIONS.teacher#, course#
    FROM TEACHERS, SECTIONS;
```

could also be expressed as

```
SELECT teacher_name, TEACHERS.teacher#, SECTIONS.teacher#, course#
    FROM TEACHERS
    CROSS JOIN SECTIONS;
```