CHAPTER 13

Joining Tables

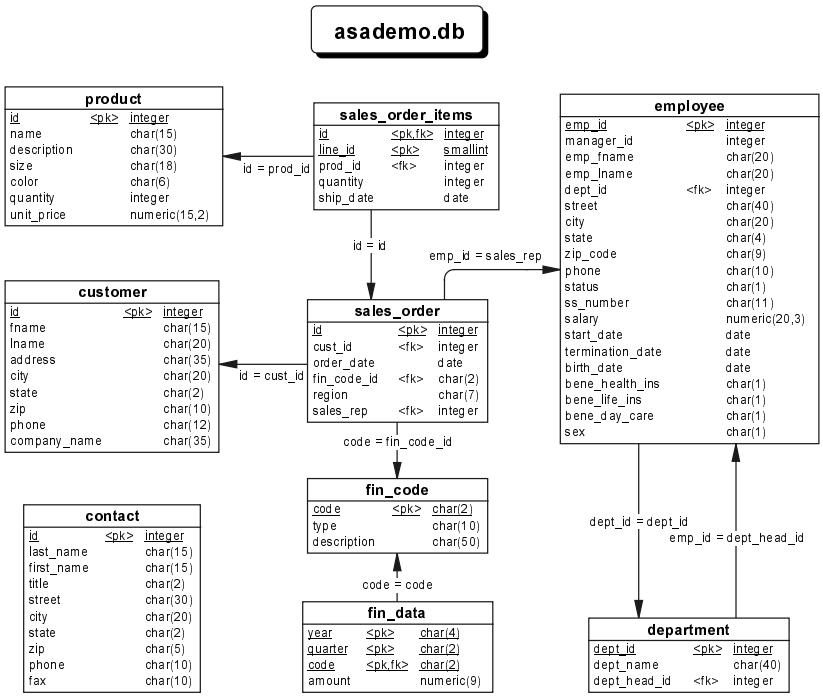
About this chapter

This chapter describes database queries that look at information in more than one table. To do this, SQL provides the **JOIN** operator. There are several different ways to join tables together in queries, and this chapter describes some of the more important ones.

Contents

The sample database

This chapter demonstrates joins using the sample database, asademo.db, included with Adaptive Server Anywhere. The sample database consists of nine tables, storing information about a fictional company.



Each box in the diagram represents a table in the database. The names listed in each box are the column names for the table.

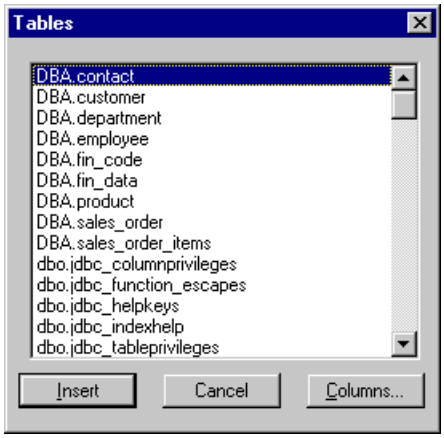
Each box contains a list of the columns in that table. The column name appears first. Next, the letters **pk** identify those columns that are part of the *primary key* and **fk** those that are part of a *foreign key*. The data type of each column appears last.

The arrows represent foreign key *relationships* between the tables.

These three new terms are defined below.

Displaying a list of tables

In Interactive SQL, you can display a list of tables by pressing the F7 key. The tables for the database are prefixed with **dba** (the owner of the tables).



The cursor keys can be used to scroll through the list of tables. Each table in the list is prefixed with a user name. This prefix is the ID of the user that created the table—the owner of the table.

Positioning the highlight on a particular table and pressing the **Columns** button displays the list of columns for that table. The ESCAPE key takes you back to the table list and pressing it again will take you back to the command window. Pressing ENTER instead of ESCAPE copies the highlighted table or column name into the command window at the current cursor position.

Press ESCAPE to leave the list.

Joining tables with the cross product

One of the tables in the sample database is **sales_order**, which lists the orders placed to the company. Each order has a **sales_rep** column, containing the employee ID of the sales representative responsible for the order. There are 648 rows in the **sales_order** table.

You can get information from two tables at the same time by listing both tables in the FROM clause of a SELECT query.

Example

The following example lists all the data in the **employee** table and the **sales_order** table:

```
SELECT *
FROM sales_order CROSS JOIN employee
```

The results of this query, displayed in the Interactive SQL data window, match every row in the **employee** table with every row in the **sales_order** table. Since there are 75 rows in the **employee** table and 648 rows in the **sales_order** table, there are 75 × 648 = 48,600 rows in the result of the join. Each row consists of all columns from the **sales_order** table followed by all columns from the **employee** table. This join is called a **full cross product**.

The cross product join is a simple starting point for understanding joins, but not very useful in itself.

Restricting a join

The most natural way to make a join useful is to insist that the **sales_rep** in the **sales_order** table be the same as the one in the **employee** table in every row of the result. Then each row contains information about an order and the sales rep responsible for it.

Example 1

To do this, add a ON phrase to the previous query to show the list of employees and their course registrations:

```
SELECT *
FROM sales_order JOIN employee
   ON sales_order.sales_rep = employee.emp_id
```

The table name is given as a prefix to identify the columns. Although not strictly required in this case, using the table name prefix clarifies the statement, and is required when two tables have a column with the same name. A table name used in this context is called a **qualifier**.

The results of this query contain only 648 rows (one for each row in the **sales_order** table). Of the original 48,600 rows in the join, only 648 of them have the employee number equal in the two tables.

Example 2

The following query is a modified version that fetches only some of the columns and orders the results.

```
SELECT employee.emp_lname, sales_order.id,
   sales_order.order_date
FROM sales_order JOIN employee
   ON sales_order.sales_rep = employee.emp_id
ORDER BY employee.emp_lname
```

If there are many tables in a SELECT command, you may need to type several qualifier names. This typing can be reduced by using a *correlation name*.

Correlation names

A **correlation name** is an alias for a particular instance of a table. This alias is valid only within a single statement. You create a correlation name by putting a short form for a table name immediately after the table name, separated by the word AS. You then *must* use the short form as a **qualifier** instead of the corresponding table name.

```
SELECT e.emp_lname, s.id, s.order_date
FROM sales_order AS s JOIN employee AS e
   ON s.sales_rep = e.emp_id
ORDER BY e.emp_lname
```

Here, two correlation names **s** and **e** are created for the **sales_order** and **employee** tables.

If you change **e.emp_id** back to **employee.emp_id**, Adaptive Server Anywhere reports an error. If you make a correlation name for an instance of a table, you *must* use that correlation name when qualifying which table a column is from; you cannot use the original table name anymore.

For a further application of correlation names, see "Self joins" on page 249.

Self joins

Correlation names, introduced in the previous section, "Restricting a join" on page 247, are also necessary when joining a single table to itself. In such a situation, your statement contains two separate **instances**, or copies, of the same table. You can only identify a particular instance by means of a correlation name.

As an example, you might wish to create a list that gives the name of each employee's manager. You can only accomplish this task by joining the **employee** table to itself.

```
SELECT e.emp_fname, e.emp_lname,
      m.emp_fname, m.emp_lname
FROM employee AS e JOIN employee AS m
   ON e.manager_id = m.emp_id
ORDER BY e.emp_lname, e.emp_fname
```

Because this statement includes two copies of the **employee** table, you *must* use correlation names to tell them apart. The above command assigns the correlation names **e** and **m** to these two copies, respectively. The join condition is that the **manager_id** value for an employee in instance **e** of the employee table is equal to the **employee_id** in instance **m** of the table.

emp_fname	enp_lname	emp_fname	enp_lname
Alex	Ahmed	Scott	Evans
Joseph	Barker	Jose	Martinez
Irene	Barletta	Scott	Evans
Jeannette	Bertrand	Jose	Martinez
Janet	Bigelow	Mary Anne	Shea
Barbara	Blaikie	Scott	Evans
Jane	Braun	Jose	Martinez

How tables are related

In order to understand how to construct other kinds of joins, you must first understand how the information in one table is related to that in another.

The **primary key** for a table identifies each row in the table. Tables are related to each other using a **foreign key**.

This section shows how primary and foreign keys together let you construct queries from more than one table.

Rows are identified by a primary key

Every table in the employee database has a **primary key**. A primary key is one or more columns that uniquely identify a row in the table. For example, an employee number uniquely identifies an employee—**emp_id** is the primary key of the employee table.

The **sales_order_items** table is an example of a table with two columns that make up the primary key. The order ID by itself does not uniquely identify a row in the **sales_order_items** table because there can be several items in an order. Also, the **line_id** number does not uniquely identify a row in the **sales_order_items** table. Both the order **ID** name and **line_id** are required to uniquely identify a row in the **sales_order_items** table. The primary key of the table is both columns taken together.

Tables are related by a foreign key

There are several tables in the employee database that refer to other tables in the database. For example, the **sales_order** table has a **sales_rep** column to indicate which employee is responsible for an order. Only enough information to uniquely identify an employee is kept in the **sales_order** table. The **sales_rep** column in the **sales_order** table is a **foreign key** to the **employee** table.

Foreign key

A **foreign key** is one or more columns that contain primary key values from another table. Each foreign key relationship in the employee database is represented by an arrow between two tables. The arrow starts at the foreign key side of the relationship and points to the primary key side of the relationship.

Join operators

Many common joins are between two tables related by a foreign key. The most common join restricts foreign key values to be equal to primary key values. The example you have already seen restricts foreign key values in the **sales_order** table to be equal to the primary key values in the **employee** table.

```
SELECT emp_lname, id, order_date
FROM sales_order JOIN employee
   ON sales_order.sales_rep = employee.emp_id
```

The query can be more simply expressed using a KEY JOIN.

Joining tables using key joins

Key joins are the best way to join two tables related by a single foreign key. For example,

```
SELECT emp_lname, id, order_date
FROM sales_order
   KEY JOIN employee
```

gives the same results as a query with a ON phrase that equates the two employee number columns:

```
SELECT emp_lname, id, order_date
FROM sales_order JOIN employee
   ON sales_order.sales_rep = employee.emp_id
```

The join operator KEY JOIN is just a short cut for typing the ON phrase; the two queries are identical.

If you look at the diagram of the employee database, foreign keys are represented by lines between tables. Anywhere that two tables are joined by a line in the diagram, you can use the KEY JOIN operator.

Joining two or more tables

Two or more tables can be joined using join operators. The following query uses four tables to list the total value of the orders placed by each customer. It connects the four tables **customer**, **sales_order**, **sales_order_items** and **product** using the single foreign-key relationships between each pair of these tables.

```
SELECT company_name,
   CAST( SUM(sales_order_items.quantity *
         product.unit_price) AS INTEGER ) AS value
FROM customer
   KEY JOIN sales_order
   KEY JOIN sales_order_items
   KEY JOIN product
GROUP BY company_name
```

company_name	value
Able Inc.	6120
AMF Corp.	3624
Amo & Sons	3216
Amy's Silk Screening	2028
Avco Ent	1752
…	…

The CAST function used in this query converts the data type of an expression. In this example the sum that is returned as an integer is converted to a value.

Joining tables using natural joins

The NATURAL JOIN operator joins two tables based on common column names. In other words, Adaptive Server Anywhere generates a ON phrase that equates the common columns from each table.

Example

For example, for the following query:

```
SELECT emp_lname, dept_name
FROM employee
   NATURAL JOIN department
```

the database engine looks at the two tables and determines that the only column name they have in common is **dept_id**. The following ON phrase is internally generated and used to perform the join:

```
FROM employee JOIN department
   ON employee.dept_id = department.dept_id
```

Errors using NATURAL JOIN

This join operator can cause problems by equating columns you may not intend to be equated. For example, the following query generates unwanted results:

```
SELECT *
FROM sales_order
NATURAL JOIN customer
```

The result of this query has no rows.

The database engine internally generates the following ON phrase:

```
FROM sales_order JOIN customer
   ON sales_order.id = customer.id
```

The **id** column in the **sales_order** table is an ID number for the *order*. The **ID** column in the **customer** table is an ID number for the *customer*. None of them matched. Of course, even if a match were found, it would be a meaningless one.

You should be careful not to use join operators blindly. Always remember that the join operator just saves you from typing the ON phrase for a foreign key or common column names. You should be conscious of the ON phrase, or you may be creating queries that give results other than what you intend.