C H A P T E R 6

Joins: Retrieving Data from Several Tables

About this chapter

When you create a database, you normalize the data by placing information specific to different objects in different tables, rather than one large table with many redundant entries.

A join operation recreates a larger table using the information from two or more tables (or views). By using different joins, you can construct a variety of these virtual tables, each suited to a particular task.

Before your start

This chapter assumes some knowledge of queries and the syntax of the select statement. Information about queries is located in "Queries: Selecting Data from a Table" on page 85. You may also wish to review the introductory material on joins, located in "Joining Tables" on page 243 of the book *First Guide to SQL Anywhere Studio*.

Contents

How joins work

A relational database stores information about different types of objects in different tables. For example, you should store information particular to employees in one table, and information that pertains to departments in another. The employee table contains information such as an employee's name and address. The department table contains information about one department, such as the name of the department and who is the department head.

Most questions can only be answered using a combination of information from the various tables; for example, the question "Who manages the Sales department?" To find the name of this person, you must identify the correct person using information from the department table, then look up that person's name in the employee table.

Joins are a means of answering such questions by forming a new virtual table that includes information from multiple tables. For example, you could create a list of the department heads by combining the information contained in the employee table and the department table. You specify which tables contain the information you need using the FROM clause.

To make the join useful, you must combine the correct columns of each table. To list department heads, each row of the combined table should contain the name of a department and the name of the employee who manages it. You control how columns are matched in the composite table either by specifying a particular type of join operation or by using the ON phrase.

Joins and the relational model

The join operation is the hallmark of the relational model of database management. More than any other feature, the join distinguishes relational database management systems from other types of database management systems.

In structured database management systems, often known as network and hierarchical systems, relationships between data values are predefined. Once a database has been set up, it is difficult to make queries about unanticipated relationships among the data.

In a relational database management system, on the other hand, relationships among data values are left unstated in the definition of a database. They become explicit when the data is manipulated: when you query the database, not when you create it. You can ask any question that comes to mind about the data stored in the database, regardless of what was intended when the database was set up.

According to the rules of good database design, called normalization rules, each table should describe one kind of entity—a person, place, event, or thing. That is why, when you want to compare information about two or more kinds of entities, you need the join operation. Relationships among data stored in different tables are discovered by joining them.

A corollary of this rule is that the join operation gives you unlimited flexibility in adding new kinds of data to your database. You can always create a new table that contains data about a different kind of entity. If the new table has a field with values similar to those in some field of an existing table or tables, it can be linked to those other tables by joining.

How joins are structured

A join operation may appear within a variety of statements, such as within the from clause of a select statement. The columns named after the FROM keyword are the columns to be included in the query results, in your desired order.

When two or more tables contain a column with the same name, you must qualify the column name explicitly to avoid ambiguity. For example, the **product** table and the **sales_order_items** table in the sample database both contain a column named **id**. If you wish to select either of these two columns, you need to identify the column you mean explicitly. If only one table uses a particular column name, the column name alone suffices.

```
SELECT product.id, sales_order_items.id, size
FROM …
```

You do not have to qualify the column name **size** because there is no ambiguity about the table to which it belongs—although these qualifiers often make your statement clearer, so it is a good idea to get in the habit of including them.

As in any select statement, column names in the select list and table names in the FROM clause must be separated by commas.

For information about queries use a single table, see "Queries: Selecting Data from a Table" on page 85.

The FROM clause

Use the FROM clause to specify which tables and views to join. You can name any two or more tables or views.

Join operators

Adaptive Server Anywhere provides four join operations:

- key joins
- natural joins
- joins using a condition, such as equality
- cross joins

Key joins, natural joins and joins on a condition may be of type inner, left-outer, or right-outer. These join types differ in the way they treat rows that have no matching row in the other table.

Data types in join columns

The columns being joined must have the same or compatible data types. Use the convert function when comparing columns whose datatypes cannot be implicitly converted.

If the datatypes used in the join are compatible, Adaptive Server Anywhere automatically converts them. For example, Anywhere converts among any of the numeric type columns, such as INT or FLOAT, and among any of the character type and date columns, such as CHAR or VARCHAR.

For the details of datatype conversions, see "Data type conversions" on page 254 of the book *Adaptive Server Anywhere Reference Manual*.

Key joins

The simplest way to join tables is to connect them using the foreign key relationships built into the database. This method is particularly economical in syntax and especially efficient.

Answer the question, "Which orders has Beth Reiser placed?"

```
SELECT customer.fname, customer.lname,
       sales_order.id, sales_order.order_date
FROM customer KEY JOIN sales_order
WHERE     customer.fname = 'Beth'
   AND    customer.lname = 'Reiser'
```

fname	lname	id	order_date
Beth	Reiser	2142	1993-01-22
Beth	Reiser	2318	1993-09-04
Beth	Reiser	2338	1993-09-24
Beth	Reiser	2449	1993-12-14
Beth	Reiser	2562	1994-03-17
Beth	Reiser	2585	1994-04-08
Beth	Reiser	2002	1993-03-20

When the option to use a key join is available, it’s generally a good idea to use it as opposed to another type.

A key join is valid if and only if exactly one foreign key is identified between the two tables. Otherwise, an error indicating the ambiguity is reported. Some constraints on these joins mean that they will not always be an available option.

- A foreign-key relationship must exist in the database. You cannot use a key join to join two tables that are not related through a foreign key.
- Only one foreign key relationship can exist between the two tables. If more than one such relationship exists, Adaptive Server Anywhere cannot decide which relationship to use and will generate an error indicating the ambiguity. You cannot specify the suitable foreign key in your statement since the syntax of the SQL language does not provide a means to do so.
- A suitable foreign key relationship must exist. You may need to create a join using two particular columns. A foreign-key relationship between the two tables may not suite your purpose.

Key joins are the default

Key join is the default join type in Adaptive Server Anywhere. Anywhere performs a key join if you do not specify the type of join explicitly, using a keyword such as KEY or NATURAL, or by including an ON phrase.

For example, Adaptive Server Anywhere performs a key join when it encounters the following statement.

```
SELECT *
FROM product JOIN sales_order_items
```

Similarly, the following join fails because there are two foreign key relationships between these tables.

```
SELECT *
FROM employee JOIN department
```

Natural joins

A **natural join** matches the rows from two tables by comparing the values from columns, one in each table, that have the same name. It restricts the results by comparing the values of columns in the two tables with the same column name. An error is reported if there is no common column name.

For example, you can join the employee and department tables using a natural join because they have only one column name in common.

```
SELECT emp_fname, emp_lname, dept_name
FROM employee NATURAL JOIN department
ORDER BY dept_name, emp_lname, emp_fname
```

emp_fname	emp_lname	dept_name
Janet	Bigelow	Finance
Kristen	Coe	Finance
James	Coleman	Finance
Jo Ann	Davidson	Finance
Denis	Higgins	Finance
Julie	Jordan	Finance
John	Letiecq	Finance
Jennifer	Litton	Finance
Mary Anne	Shea	Finance
Alex	Ahmed	Marketing
Irene	Barletta	Marketing
Barbara	Blaikie	Marketing

Column names such as **description** or **address** often cause a NATURAL JOIN to return different results than expected. Another means of specifying a join is by means of a condition, supplied within an ON phrase. Here, you have a wide number of options at your disposal.

Joins using comparisons

You can specify a **join condition** for any join type except CROSS JOIN. Alternatively you can create a join using a condition instead of a keyword, instead of using a KEY or NATURAL JOIN. You specify a join condition by inserting an ON phrase immediately adjacent to the join to which it applies.

Natural joins and key joins use **generated join conditions**; that is, the keyword KEY or NATURAL indicates a restriction on the join results.

For a natural join, the generated join condition is based on the names of columns in the tables being joined; for a key join, the condition is based on a foreign key relationship between the two tables.

In the sample database, the following are logically equivalent:

```
SELECT *
FROM sales_order JOIN customer
   ON sales_order.cust_id = customer.id
```

```
SELECT *
FROM sales_order KEY JOIN customer
```

The following two are also equivalent:

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

```
SELECT *
FROM department NATURAL JOIN employee
```

When you join two tables, the columns you compare must have the same or similar data types.

Join types

There are several types of joins, such as equijoins, natural joins, and outer joins. The most common join, the equijoin, is based on equality. The following command lists each order number and the name of the customer who placed them.

```
SELECT sales_order.id, customer.fname, customer.lname
FROM sales_order JOIN customer
   ON sales_order.cust_id = customer.id
```

The condition for joining the values in two columns does not need to be equality (=). You can use any of the other comparison operators: not equal (<>), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=).

Using the WHERE clause in join statements

You can use the WHERE clause to determine which rows are included in the results. In this role, it acts exactly like it does when using a single table, selecting only the rows that interest you.

The WHERE clause can also specify the connection between the tables and views named in the FROM clause. In this role, it acts somewhat like the ON phrase. In fact in the case of inner joins, the behavior is identical. However, in outer joins, for example, the same condition can produce different results if moved from an ON phrase to the WHERE clause because null values are treated differently in these two contexts. The ON phrase allows you to isolate the join constraints and can make your join statement easier to read.

Inner, left-outer, and right-outer joins

Inner joins and outer joins differ in their treatment of rows that have no match in the other table: rows appear in an inner join only if both tables contain at least one row that satisfies the join condition.

Because inner joins are the default, you do not need to specify the INNER keyword explicitly. Should you wish to use it for clarity, place it immediately before the JOIN keyword.

For example, each row of

```
SELECT fname, lname, order_date
FROM customer
   KEY INNER JOIN sales_order
ORDER BY order_date
```

contains the information from one **customer** row and one **sales_order** row. If a particular customer has placed no orders, the join will contain no information about that customer.

fname	lname	order_date
Hardy	Mums	1993-01-02
Tommie	Wooten	1993-01-03
Aram	Najarian	1993-01-03
Alfredo	Margolis	1993-01-06
Elmo	Smythe	1993-01-06
Malcolm	Naddem	1993-01-07

Because inner joins are the default, you obtain the same result using the following clause.

```
FROM customer JOIN sales_order
```

By contrast, an **outer join** contains rows whether or not a row appears in the opposite table to satisfy the join condition. Use the keywords LEFT or RIGHT to identify the table that is to appear in its entirety.

- ♦ A LEFT OUTER JOIN contains every row in the *left*-hand table.
- ♦ A RIGHT OUTER JOIN contains every row in the *right*-hand table.

For example, the outer join

```
SELECT fname, lname, order_date
FROM customer
   KEY LEFT OUTER JOIN sales_order
ORDER BY order_date
```

includes all customers, whether or not they have placed an order. If a particular customer has placed no orders, each column in the join corresponding to order information will contain the NULL value.

fname	lname	order_date
Lewis N.	Clark	(NULL)
Jack	Johnson	(NULL)
Jane	Doe	(NULL)
John	Glenn	(NULL)
Dominic	Johansen	(NULL)
Stanley	Jue	(NULL)
Harry	Jones	(NULL)
Marie	Curie	(NULL)
Elizibeth	Bordon	(NULL)
Len	Manager	(NULL)
Tony	Antolini	(NULL)
Tom	Cruz	(NULL)
Janice	O Toole	(NULL)
Stevie	Nickolas	(NULL)
Philipe	Fernandez	(NULL)
Jennifer	Stutzman	(NULL)
William	Thompson	(NULL)
Hardy	Mums	1993-01-02
Tommie	Wooten	1993-01-03
Aram	Najarian	1993-01-03

The keywords INNER, LEFT OUTER, and RIGHT OUTER may appear as modifiers in key joins, natural joins, and joins that using a comparison. These modifiers do not apply to cross joins.

Outer joins and join conditions

A common mistake is to place a join condition, which should appear in an ON phrase, in a WHERE clause. Here, the same condition often produces different results. This difference is best explained through a conceptual explanation of the way that Adaptive Server Anywhere processes a select statement.

1 First, Adaptive Server Anywhere logically completes all joins. When doing so, it uses only conditions placed within an ON phrase. When the values in one table are missing or null-valued, the behavior depends upon the type of join: inner, left-outer, or right-outer.

2 Once the join is complete, Adaptive Server Anywhere logically deletes those rows for which the condition within the WHERE clause evaluates to either FALSE or UNKNOWN.

Because conditions are treated differently, the effect of moving a condition from an ON phrase to a WHERE clause is usually to convert the join to an inner join, regardless of the type of join specified.

With INNER JOINS, specifying a join condition is equivalent to adding the join condition to the WHERE clause. However, the same is not true for OUTER JOINS.

For example, the following statement causes a left-outer join.

```
SELECT *
FROM customer LEFT OUTER JOIN sales_order
   ON customer.id = sales_order.cust_id
```

In contrast, the following two statements both create inner joins and select the same set of rows.

```
SELECT *
FROM customer LEFT OUTER JOIN sales_order
   WHERE customer.id = sales_order.cust_id
```

```
SELECT *
FROM customer INNER JOIN sales_order
   ON customer.id = sales_order.cust_id
```

The first of these two statements can be thought of as follows: First, left-outer join the customer table to the **sales_order** table. For those customers who have not yet placed an order, fill the sales order fields with nulls. Next, select those rows in which the customer id values are equal. For those customers who have not placed orders, these values will be NULL. Since comparing any value to NULL results in the special value UNKOWN, these rows are eliminated and the statement reduces to an inner join.

This methodology describes the logical effect of the statements you type, not how Adaptive Server Anywhere goes about processing them. For further information, see "How joins are processed" on page 149.

Self-joins and correlation names

Joins can compare values within the same column, or two different columns of a *single* table. These joins are called self-joins. For example, you can create a list all the employees and the name of each person's manager by joining the employee table to itself.

In such a join, you cannot distinguish the columns by the conventional means because the join will contain two copies of every column.

For example, suppose you want to create a table of employees that includes the names of their managers. The following query does *not* answer this question.

```
SELECT *
FROM employee JOIN employee
   ON employee.manager_id = employee.emp_id
```

In fact, this statement is *semantically equivalent* to the much simpler statement below.

```
SELECT *
FROM employee
```

When constructing joins, Adaptive Server Anywhere treats all tables or views that have the same name as the same instance of a view or table. The name **employee** is treated as only one instance of the employee table because the same name appears at both locations in the FROM clause.

Use correlation names to distinguish instances of a table

To distinguish an individual instance of a table, use a **correlation name**. A correlation name is an alias for an instance of a table or view. You define a correlation name in the FROM clause. Once defined, you *must* use the correlation name in place of the table name elsewhere within your statement, including the selection list, wherever you refer to that instance of the table.

The following statement uses the correlation names **report** and **manager** to distinguish the two instances of the employee table and so correctly creates the list of employees and their managers.

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

This statement produces the result shown below. The employee names appear in the two left-hand columns and the names of their managers on the right.

emp_fname	emp_lname	emp_fname	emp_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
Robert	Breault	David	Scott
Matthew	Bucceri	Scott	Evans
Joyce	Butterfield	Scott	Evans

Using correlation names

Choose short, concise correlation names to make your statements easier to read. In many cases, names only one or two characters in length will suffice.

While you *must* use correlation names for a self-join to distinguish multiple instances of a table, they can make many other statements more readable too. For example, the statement

```
SELECT customer.fname, customer.lname,
       sales_order.id, sales_order.order_date
FROM customer KEY JOIN sales_order
WHERE    customer.fname = 'Beth'
   AND   customer.lname = 'Reiser'
```

becomes more compact if you use the correlation name **c** for **customer** and **so** for **sales_order**:

```
SELECT c.fname, c.lname, so.id, so.order_date
FROM customer AS c KEY JOIN sales_order AS so
WHERE    c.fname = 'Beth'
   AND   c.lname = 'Reiser'
```

For brevity, you can even eliminate the keyword AS. It is redundant because the syntax of the SQL language identifies the correlation names: they are separated from the corresponding table name by only a space, not a comma.

```
SELECT c.fname, c.lname, so.id, so.order_date
FROM customer c KEY JOIN sales_order so
WHERE    c.fname = 'Beth'
   AND   c.lname = 'Reiser'
```

For further details of the rules regarding correlation names and instances of a table within a FROM clause, see "Joining more than two tables" on page 151.

Cross joins

As for other types of joins, each row in a cross join is a combination of one column from the first table and one column from the second table. Unlike other joins, a cross join contains no restrictions. All possible combinations of rows are present.

Each row of the first table appears exactly once with each row of the second table. Hence, the number of rows in the join is the product of the number of rows in the individual tables.

Inner and outer modifiers do not apply to cross joins

Except in the presence of additional restrictions, all rows of both tables always appear in the result. Thus, the keywords INNER, LEFT OUTER and RIGHT OUTER are not applicable to cross joins.

The query

```
SELECT *
FROM table1 CROSS JOIN table2
```

has a result set as follows:

- As long as **table1** is not the same name as **table2**:
    - The result set includes all columns in **table1** and all columns in **table2**.
    - There is one row in the result set for each combination of a row in **table1** and a row in **table2**. If **table1** has *n1* rows and **table2** has *n2* rows, the query returns *n1* x *n2* rows.
- If **table1** is the same table as **table2**, and neither is given a correlation name, the result set is simply the rows of **table1**.

Self-joins and cross joins

The following self-join produces a list of pairs of employees. Each employee names appears in combination with every employee name.

```
SELECT a.emp_fname, a.emp_lname,
      b.emp_fname, b.emp_lname
FROM employee AS a CROSS JOIN employee AS b
```

emp_fname	emp_lname	Emp_fname	emp_lname
Fran	Whitney	Fran	Whitney
Matthew	Cobb	Fran	Whitney
Philip	Chin	Fran	Whitney
Julie	Jordan	Fran	Whitney
Robert	Breault	Fran	Whitney
Melissa	Espinoza	Fran	Whitney
Jeannette	Bertrand	Fran	Whitney

Since the employee table has 75 rows, this join contains 75 x 75 = 5625 rows. It includes, however, rows which that list each employee with themselves. For example, it contains the row

emp_fname	emp_lname	emp_fname	emp_lname
Fran	Whitney	Fran	Whitney

To remove that list the same employee's name twice, use the following command.

```
SELECT a.emp_fname, a.emp_lname,
       b.emp_fname, b.emp_lname
FROM employee AS a CROSS JOIN employee AS b
WHERE a.emp_lname <> b.emp_lname
   OR a.emp_fname <> b.emp_fname
```

Without these rows, the join contains 75 x 74 = 5550 rows.

This join contains rows that pair each employee with every other employee, but because each pair of names can appear in two possible orders, each pair appears twice. For example, the result of the above join contains the following two rows.

emp_fname	emp_lname	emp_fname	emp_lname
Matthew	Cobb	Fran	Whitney
Fran	Whitney	Matthew	Cobb

If the order of the names is not important, you can produce a list of the (75 x 74)/2! = 2775 unique pairs.

```
SELECT a.emp_fname, a.emp_lname,
      b.emp_fname, b.emp_lname
FROM employee AS a CROSS JOIN employee AS b
WHERE a.emp_lname < b.emp_lname
   OR (a.emp_lname = b.emp_lname
   AND a.emp_fname < b.emp_fname)
```

This statement eliminates duplicate lines by selecting only those rows in which the name of employee **a** is alphabetically less than that of employee **b**.

For more information, see "Self-joins and correlation names" on page 143.

How joins are processed

Knowing how joins are processed helps to understand them—and to figure out why, when you incorrectly state a join, you sometimes get unexpected results. This section describes the processing of joins in conceptual terms. When executing your statements, Adaptive Server Anywhere uses a sophisticated strategy to obtain the same results by more efficient means.

1. The first logical step in processing a join is to use the join condition to form the Cartesian product of the tables—all the possible combinations of the rows from each of the tables. The number of rows in a Cartesian product of tables is the product of the number of rows in the individual tables. This Cartesian product contains all the rows that satisfy your join condition, and all of the columns from all of the tables.
2. The next logical step is to select the rows you want using conditions in the WHERE clause. Whereas you may include NULL values for missing rows using a left- or right-outer join, Adaptive Server Anywhere selects rows only if the condition evaluates to TRUE. It omits rows if the condition evaluates to either FALSE or UNKNOWN.
3. If you include a GROUP BY clause, the rows are partitioned according to your conditions. Next, rows are selected from these partitions according to any conditions in the HAVING clause.
4. If the statement includes an ORDER BY clause, then Adaptive Server Anywhere uses it to order the remaining rows. When you do not specify an ordering, make no assumptions regarding the order of the rows.
5. Finally, Adaptive Server Anywhere returns those columns you specified in your select statement.

Tips
Adaptive Server Anywhere accepts a wide range of syntax. This flexibility means that most queries result in an answer, but sometimes not the one you intended. The following precautions will help you avoid this peril.

1. Always use correlation names.
2. Try eliminating a WHERE clause when testing a new statement.
3. Avoid mixing inner joins with left-outer or right-outer joins.
4. Examine the plan for your query—does it include all the tables?

Performance considerations

Generally, Adaptive Server Anywhere prefers to process joins by selecting information in one table, then performing an indexed look-up to get the rows it needs from another. Anywhere carefully optimizes each of your statements before executing it. As long as your statement correctly identifies the information you want, it usually doesn't matter what syntax you use.

In particular, Adaptive Server Anywhere is free to reconstruct your statement to any form that is semantically equivalent. It will almost always will do so, so as to compute your result efficiently. You can determine the result of a statement using the above methods, but Anywhere usually obtains the result by another means.

One means by which Adaptive Server Anywhere improves performance is to use indexes whenever doing so will improve performance. Columns that are part of a primary or secondary key are indexed automatically. Other columns are not. Creating additional indexes on columns involved in a join, either as part of a join condition or in a where clause, can improve performance dramatically.

For further performance tips, see "Monitoring and Improving Performance" on page 623.

Joining more than two tables

To carry out many queries, you need you will need to join more than two tables. Here, you have two options at your disposal.

The first method to join multiple tables. The following command answers the question "What items were are listed on order number 2015?"

```
SELECT product.name, size, sales_order_items.quantity
FROM sales_order
   KEY JOIN sales_order_items
   KEY JOIN product
WHERE sales_order.id = 2015
```

id	name	size	quantity
300	Tee Shirt	Small	24
301	Tee Shirt	Medium	24
302	Tee Shirt	One size fits all	24
700	Shorts	Medium	24

When you want to join a number of tables sequentially, the above syntax makes a lot of sense. However, sometimes you need to join a single table to several others that surround it.

Star joins

Some joins must join a single table to several others around it. This type of join is called a **star join**.

As an example, create a list the names of the customers that have placed orders with Rollin Overbey.

```
SELECT c.fname, c.lname, o.order_date
FROM sales_order AS o KEY JOIN customer AS c,
   sales_order AS o KEY JOIN employee AS e
WHERE e.emp_fname = 'Rollin' AND e.emp_lname = 'Overbey'
ORDER BY o.order_date
```

Notice that one of the tables in the FROM clause, **employee**, does not contribute any columns to the results. Nor do any of the columns that are joined—customer id and employee id—appear in the results. Nonetheless, this join is possible only by using the **employee** table.

fname	lname	order_date
Tommie	Wooten	1993-01-03
Michael	Agliori	1993-01-08
Salton	Pepper	1993-01-17
Tommie	Wooten	1993-01-23
Michael	Agliori	1993-01-24

The following statement uses a star join around the **sales_order** table. The result is a list that shows all the customers and the total quantity of each type of product that they have ordered. Some customers have not placed orders, so the other values for these customers are NULL. In addition, it shows the name of the manager of the sales person through whom they placed the orders.

```
SELECT c.fname, p.name, SUM(i.quantity), m.emp_fname
FROM   sales_order o
          KEY LEFT OUTER JOIN sales_order_items i
          KEY LEFT OUTER JOIN product p,
       sales_order o
          KEY RIGHT OUTER JOIN customer c,
       sales_order o
          KEY LEFT OUTER JOIN employee e
          LEFT OUTER JOIN employee m
             ON e.manager_id = m.emp_id
WHERE c.state = 'CA'
GROUP BY c.fname, p.name, m.emp_fname
ORDER BY SUM(i.quantity) DESC, c.fname
```

Note the following details of this statement:

- The join centers on the customer table.
- The keyword AS is optional and has been omitted.
- All joins must be outer joins to keep the outer join with the customer table includes null values.
- The condition e.manager_id = m.emp_id must be placed in the ON phrase instead of the WHERE clause. The result of this statement would been inner join if this condition moved into the WHERE clause.

The statement produces the results shown in the table, below.

fname	name	SUM(i.quantity)	emp_fname
Harry	(NULL)	(NULL)	(NULL)
Jane	(NULL)	(NULL)	(NULL)
Philipe	(NULL)	(NULL)	(NULL)
Sheng	Baseball Cap	240	Moira
Laura	Tee Shirt	192	Moira
Moe	Tee Shirt	192	Moira
Leilani	Sweatshirt	132	Moira
Almen	Baseball Cap	108	Moira

Joins involving derived tables

You can nest queries within a FROM clause. Tables created in this manner are called **derived tables**. Using derived queries, you can perform grouping of groups or construct a join with a group, without having to create a view.

In the following example, the inner SELECT statement (enclosed in parentheses) creates a derived table, grouped by customer id values. The outer SELECT statement assigns this table the correlation name **sales_order_counts** and joins it to the **customer** table using a join condition.

```
SELECT lname, fname, number_of_orders
FROM customer join
   (  SELECT cust_id, count(*)
      FROM sales_order
      GROUP BY cust_id   )
   AS sales_order_counts (cust_id, number_of_orders)
   ON (customer.id = sales_order_counts.cust_id)
WHERE number_of_orders > 3
```

The result is a table of the names of those customers who have placed more than three orders, including the number of orders each has placed.

Transact-SQL outer joins

Joins that include all rows, regardless of whether or not they match the join condition, are called **outer joins**. Adaptive Server Anywhere supports both left and right outer joins via the LEFT OUTER and RIGHT OUTER keywords. For compatibility with Adaptive Server Enterprise, Anywhere supports the Transact-SQL-language counterparts of these keywords.

In the Transact-SQL dialect, joins are accomplished by separating table names with commas in the FROM clause. The join conditions appear in the WHERE clause, rather than in the ON phrase. Special conditional operators indicate the type of join.

Transact-SQL left-outer joins

The left outer join operator, *=, selects all rows from the left hand table that meet the statement's restrictions. The right hand table generates values if there is a match on the join condition. Otherwise, the second table generates null values.

For example, the following left outer join lists *all* customers and finds their order dates (if any):

```
SELECT fname, lname, order_date
FROM customer, sales_order
WHERE customer.id *= sales_order.cust_id
ORDER BY order_date
```

Preserved and null-supplying tables

A table is either a **preserved** or a **null-supplying** table for an outer join. If the join operator is *=, the second table is the null-supplying table; if the join operator is =*, the first table is the null-supplying table.

You can compare a column from the inner table to a constant as well as using it in the outer join. For example, you can use the following statement to find information about customers in California.

```
SELECT fname, lname, order_date
FROM customer, sales_order
WHERE customer.state = 'CA'
   AND customer.id *= sales_order.cust_id
ORDER BY order_date
```

However, the inner table in an outer join cannot also participate in a regular join clause.

Bit columns
Since bit columns do not permit null values, a value of 0 appears in an outer join when there is no match for a bit column that is in the inner table.

Transact-SQL right-outer joins

The right outer join, =*, selects all rows from the second table that meet the statement's restrictions. The first table generates values if there is a match on the join condition. Otherwise, the first table generates null values.

The right outer join is specified with the comparison operator =*, which indicates that all the rows in the second table are to be included in the results, regardless of whether there is matching data in the first table.

Substituting this operator in the outer join query shown earlier gives this result:

```
SELECT fname, lname, order_date
FROM sales_order, customer
WHERE sales_order.cust_id =* customer.id
ORDER BY order_date
```

You can further restrict an outer join by comparing it to a constant. This means that you can zoom in on precisely the values you really want to see and use the outer join to list the rows that did not make the cut.

Transact-SQL outer join restrictions

There are several restrictions for Transact-SQL outer joins:

- You cannot mix SQL/92 syntax and Transact-SQL outer join syntax in a single query. This applies to views used by a query also: if a view is defined using one dialect for an outer join, the same dialect must be used for any outer-join queries on that view.
- A table cannot participate in both a Transact-SQL outer join clause and a regular join clause. For example, the following WHERE clause is not allowed:

  ```
  WHERE R.x *= S.x
  AND S.y = T.y
  ```

  When you cannot rewrite your query to avoid using a table in both an outer join and a regular join clause, you must divide your statement into two separate queries.

- You cannot use a subquery that contains the null-supplying table of an outer join. For example, the following WHERE clause is not allowed:

  ```
  WHERE R.x *= S.y
  AND EXISTS ( SELECT *
               FROM T
               WHERE T.x = S.x )
  ```

- If you submit a query with an outer join and a qualification on a column from the null-supplying table of the outer join, the results may not be what you expect. The qualification in the query does not restrict the number of rows returned, but rather affects which rows contain the null value. For rows that do not meet the qualification, a null value appears in the inner table's columns of those rows.

Views used with Transact-SQL outer joins

If you define a view with an outer join, and then query the view with a qualification on a column from the inner table of the outer join, the results may not be what you expect. The query returns all rows from the null-supplying table. Rows that do not meet the qualification show a NULL value in the appropriate columns of those rows.

The following rules determine what types of updates you can make to columns through join views:

- DELETE statements are not allowed on join views.
- INSERT statements are not allowed on join views created WITH CHECK OPTION.
- UPDATE statements are allowed on join views WITH CHECK OPTION. The update fails if any of the affected columns appears in the WHERE clause, in an expression that includes columns from more than one table.
- If you insert or update a row through a join view, all affected columns must belong to the same base table.

How NULL affects Transact-SQL joins

NULL values in tables or views being joined will never match each other. Since bit columns do not permit NULLs, a value of 0 appears in an outer join when there is no match for a bit column that is in the inner table.

The result of a join of NULL with any other value is NULL. Because null values represent unknown or inapplicable values, Transact-SQL has no reason to believe that one unknown value matches another.

You can detect the presence of null values in a column from one of the tables being joined only by using an outer join. If there are two tables, each of which has a NULL in the column that will participate in the join. A left outer join displays the NULL in the first table.