

-- 02 JOIN

-- test.db

-- join example tables, on\_left and on\_right

CREATE TABLE on\_left ( id INTEGER, description TEXT );

CREATE TABLE on\_right ( id INTEGER, description TEXT );

INSERT INTO on\_left VALUES ( 1, 'on\_left 01' );

INSERT INTO on\_left VALUES ( 2, 'on\_left 02' );

INSERT INTO on\_left VALUES ( 3, 'on\_left 03' );

INSERT INTO on\_left VALUES ( 4, 'on\_left 04' );

INSERT INTO on\_left VALUES ( 5, 'on\_left 05' );

INSERT INTO on\_left VALUES ( 6, 'on\_left 06' );

INSERT INTO on\_left VALUES ( 7, 'on\_left 07' );

INSERT INTO on\_left VALUES ( 8, 'on\_left 08' );

INSERT INTO on\_left VALUES ( 9, 'on\_left 09' );

INSERT INTO on\_right VALUES ( 6, 'on\_right 06' );

INSERT INTO on\_right VALUES ( 7, 'on\_right 07' );

INSERT INTO on\_right VALUES ( 8, 'on\_right 08' );

INSERT INTO on\_right VALUES ( 9, 'on\_right 09' );

INSERT INTO on\_right VALUES ( 10, 'on\_right 10' );

INSERT INTO on\_right VALUES ( 11, 'on\_right 11' );

INSERT INTO on\_right VALUES ( 11, 'on\_right 12' );

INSERT INTO on\_right VALUES ( 11, 'on\_right 13' );

INSERT INTO on\_right VALUES ( 11, 'on\_right 14' );

SELECT \* FROM on\_left;

SELECT \* FROM on\_right;

-- inner join

SELECT l.description AS on\_left, r.description AS on\_right

FROM on\_left AS l

INNER JOIN on\_right AS r ON l.id = r.id;

-- left join

SELECT l.description AS on\_left, r.description AS on\_right

FROM on\_left AS l

RIGHT OUTER JOIN on\_right AS r ON l.id = r.id;

-- right join

SELECT l.description AS on\_left, r.description AS on\_right

FROM on\_left AS l

LEFT OUTER JOIN on\_right AS r ON l.id = r.id;

## Understanding joins

It's the nature of a relational database that some tables contain information related to other tables. Using the join statement, SQL has powerful tools for extracting related data from multiple tables. Typically unique ID fields are used to create relationships. All modern database systems support automatic generation of these ID fields, and they work well for creating and managing simple or complex relationships between tables. When you need a result that includes related rows from multiple tables, you'll need to use a joined query.

It's easy to understand joins if you visualize your tables as a Venn diagram, where each of your tables are represented by intersecting shapes. The intersection of the shapes where the tables overlap, are the records where our condition is met. I.D. fields are often used for this purpose where the condition to be met is matching ID's. The simplest and most common form of a join is the inner join. This is the default and it's the join you get when you use the join key word by itself.

The result of an inner join will include rows from both tables where the joined condition is met. The inner key word is typically omitted as this is the default join. The outer join is less common, but still important to understand. A left outer join includes the rows where the condition is met, plus all the rows from the table on the left, where the condition is not met. The outer key word is typically omitted as a left join is presumed to be an outer join. Likewise, a right outer join includes all the rows from the table on the right.

This is considered a special case in many databases don't support right joins. MySQL, does support right joins. Generally a right join can be written as a left join by simply changing the order of the tables in the query. A full outer join includes all the rows from both tables, including those where the condition is met. Many database systems including MySQL do not implement full outer join. There are many variations of these basic joins implemented in different ways by the different database system vendors.

Write an INNER, LEFT, and a RIGHT JOIN.

```
SELECT * FROM on_left;  
SELECT * FROM on_right;
```

id	description	id	description
1	on_left 01	6	on_right 06
2	on_left 02	7	on_right 07
3	on_left 03	8	on_right 08
4	on_left 04	9	on_right 09
5	on_left 05	10	on_right 10
6	on_left 06	11	on_right 11
7	on_left 07	11	on_right 12
8	on_left 08	11	on_right 13
9	on_left 09	11	on_right 14

-- inner join

```
SELECT l.description AS on_left, r.description AS on_right
FROM on_left AS l
INNER JOIN on_right AS r ON l.id = r.id;
```

-- left join

```
SELECT l.description AS on_left, r.description AS on_right
FROM on_left AS l
RIGHT OUTER JOIN on_right AS r ON l.id = r.id;
```

-- right join

```
SELECT l.description AS on_left, r.description AS on_right
FROM on_left AS l
LEFT OUTER JOIN on_right AS r ON l.id = r.id;
```

The inner join will show only results where the condition is met. So the condition l.id equals r.id, is true for the rows where the id is equal in the two tables. So it's true for this on\_left id six and this on\_right id six, and this on\_left id seven, and this on\_right id seven. And so the query result shows these **intersections**. Columns are called left and right because I said AS on\_left and AS on\_right in the SELECT statement. And you'll notice that those are the rows where that condition is true.

<b>on_left</b>	<b>on_right</b>
on_left 06	on_right 06
on_left 07	on_right 07
on_left 08	on_right 08
on_left 09	on_right 09

LEFT JOIN results are there where the conditions met, but also all of the results from the on\_left column, where the condition is not met. The on\_right column in the result says NULL for those because there is no corresponding column in the on\_right table.

<b>on_left</b>	<b>on_right</b>
on_left 06	on_right 06
on_left 07	on_right 07
on_left 08	on_right 08
on_left 09	on_right 09
on_left 01	<i>NULL</i>
on_left 02	<i>NULL</i>
on_left 03	<i>NULL</i>
on_left 04	<i>NULL</i>
on_left 05	<i>NULL</i>

RIGHT JOIN results are where the conditions met, but also all of the results from

the on\_right column, where the condition is not met. The on\_left column in the result says NULL for those because there is no corresponding column in the on\_left table.

on_left	on_right
on_left 06	on_right 06
on_left 07	on_right 07
on_left 08	on_right 08
on_left 09	on_right 09
NULL	on_right 10
NULL	on_right 11
NULL	on_right 12
NULL	on_right 13
NULL	on_right 14

# Lesson 12. Joining Tables

*In this lesson, you'll learn what joins are, why they are used, and how to create `SELECT` statements using them.*

## Understanding Joins

One of SQL's most powerful features is the capability to join tables on-the-fly within data retrieval queries. Joins are one of the most important operations that you can perform using SQL `SELECT`, and a good understanding of joins and join syntax is an extremely important part of learning SQL.

Before you can effectively use joins, you must understand relational tables and the basics of relational database design. What follows is by no means complete coverage of the subject, but it should be enough to get you up and running.

## Understanding Relational Tables

The best way to understand relational tables is to look at a real-world example.

Suppose you had a database table containing a product catalog, with each catalog item in its own row. The kind of information you would store with each item would include a product description and price, along with vendor information about the company that creates the product.

Now suppose that you had multiple catalog items created by the same vendor. Where would you store the vendor information (things like vendor name, address, and contact information)? You wouldn't want to store that data along with the products for several reasons:

- Because the vendor information is the same for each product that vendor produces, repeating the information for each product is a waste of time and storage space.
- If vendor information changes (for example, if the vendor moves or his area code changes), you would need to update every occurrence of the vendor information.
- When data is repeated, (that is, the vendor information is used with each product), there is a high likelihood that the data will not be entered exactly the same way each time. Inconsistent data is extremely difficult to use in reporting.

The key here is that having multiple occurrences of the same data is never a good thing, and that principle is the basis for relational database design. Relational tables are designed so that information is split into multiple tables, one for each data type. The tables are related to each other through common values (and thus the *relational* in relational design).

In our example, you can create two tables, one for vendor information and one for product information. The `Vendors` table contains all the vendor information, one table row per vendor, along with a unique identifier for each vendor. This value, called a *primary key*, can be a vendor ID, or any other unique value.

The `Products` table stores only product information, and no vendor specific information other than the vendor ID (the `Vendors` table's primary key). This key relates the `Vendors` table to the `Products` table, and using this vendor ID enables you to use the `Vendors` table to find the details about the appropriate vendor.

What does this do for you? Well, consider the following:

- Vendor information is never repeated, and so time and space are not wasted.
- If vendor information changes, you can update a single record, the one in the `Vendors` table. Data in related tables does not change.
- As no data is repeated, the data used is obviously consistent, making data reporting and manipulation much simpler.

The bottom line is that relational data can be stored efficiently and manipulated easily. Because of this, relational databases scale far better than non-relational databases.

### Scale

Able to handle an increasing load without failing. A well-designed database or application is said to *scale well*.

## Why Use Joins?

As just explained, breaking data into multiple tables enables more efficient storage, easier manipulation, and greater scalability. But these benefits come with a price.

If data is stored in multiple tables, how can you retrieve that data with a single `SELECT` statement?

The answer is to use a join. Simply put, a join is a mechanism used to associate tables within a `SELECT` statement (and thus the name join). Using a special syntax, multiple tables can be joined so that a single set of output is returned, and the join associates the correct rows in each table on-the-fly.

### Note: Using Interactive DBMS Tools

Understand that a join is not a physical entity—in other words, it does not exist in the actual database tables. A join is created by the DBMS as needed, and it persists for the duration of the query execution.

Many DBMSs provide graphical interfaces that can be used to define table relationships interactively. These tools can be invaluable in helping to maintain referential integrity. When using relational tables, it is important that only valid data is inserted into relational columns. Going back to the example, if an invalid vendor ID is stored in the `Products` table, those products would be inaccessible because they would not be related to any vendor. To prevent this from occurring, the database can be instructed to only allow valid values (ones present in the `Vendors` table) in the vendor ID column in the `Products` table. Referential integrity means that the DBMS enforces data integrity rules. And these rules are often managed through DBMS provided interfaces.

## Creating a Join

Creating a join is very simple. You must specify all the tables to be included and how they are related to each other. Look at the following example:

### Input

---

[Click here to view code image](#)

```
SELECT vend_name, prod_name, prod_price
FROM Vendors, Products
WHERE Vendors.vend_id = Products.vend_id;
```

## Output

[Click here to view code image](#)

vend_name	prod_name	prod_price
-----	-----	-----
Doll House Inc.	Fish bean bag toy	3.4900
Doll House Inc.	Bird bean bag toy	3.4900
Doll House Inc.	Rabbit bean bag toy	3.4900
Bears R Us	8 inch teddy bear	5.9900
Bears R Us	12 inch teddy bear	8.9900
Bears R Us	18 inch teddy bear	11.9900
Doll House Inc.	Raggedy Ann	4.9900
Fun and Games	King doll	9.4900
Fun and Games	Queen doll	9.4900

## Analysis

Let’s take a look at the preceding code. The SELECT statement starts in the same way as all the statements you’ve looked at thus far, by specifying the columns to be retrieved. The big difference here is that two of the specified columns (prod\_name and prod\_price) are in one table, whereas the other (vend\_name) is in another table.

Now look at the FROM clause. Unlike all the prior SELECT statements, this one has two tables listed in the FROM clause, Vendors and Products. These are the names of the two tables that are being joined in this SELECT statement. The tables are correctly joined with a WHERE clause that instructs the DBMS to match vend\_id in the Vendors table with vend\_id in the Products table.

You’ll notice that the columns are specified as Vendors.vend\_id and Products.vend\_id. This fully qualified column name is required here because if you just specified vend\_id, the DBMS cannot tell which vend\_id columns you are referring to. (There are two of them, one in each table.) As you can see in the preceding output, a single SELECT statement returns data from two different tables.

### Caution: Fully Qualifying Column Names

As noted in the previous lesson, you must use the fully qualified column name (table and column separated by a period) whenever there is a possible ambiguity about which column you are referring to. Most DBMSs will return an error message if you refer to an ambiguous column name without fully qualifying it with a table name.

## The Importance of the WHERE Clause

It might seem strange to use a WHERE clause to set the join relationship, but actually, there is a very good reason for this. Remember, when tables are joined in a SELECT statement, that relationship is



constructed on-the-fly. There is nothing in the database table definitions that can instruct the DBMS how to join the tables. You have to do that yourself. When you join two tables, what you are actually doing is pairing every row in the first table with every row in the second table. The `WHERE` clause acts as a filter to only include rows that match the specified filter condition—the join condition, in this case. Without the `WHERE` clause, every row in the first table will be paired with every row in the second table, regardless of if they logically go together or not.

**Cartesian Product**

The results returned by a table relationship without a join condition. The number of rows retrieved will be the number of rows in the first table multiplied by the number of rows in the second table.

To understand this, look at the following `SELECT` statement and output:

**Input**

[Click here to view code image](#)

```
SELECT vend_name, prod_name, prod_price
FROM Vendors, Products;
```

**Output**

[Click here to view code image](#)

vend_name	prod_name	prod_price
-----	-----	-----
Bears R Us	8 inch teddy bear	5.99
Bears R Us	12 inch teddy bear	8.99
Bears R Us	18 inch teddy bear	11.99
Bears R Us	Fish bean bag toy	3.49
Bears R Us	Bird bean bag toy	3.49
Bears R Us	Rabbit bean bag toy	3.49
Bears R Us	Raggedy Ann	4.99
Bears R Us	King doll	9.49
Bears R Us	Queen doll	9.49
Bear Emporium	8 inch teddy bear	5.99
Bear Emporium	12 inch teddy bear	8.99
Bear Emporium	18 inch teddy bear	11.99
Bear Emporium	Fish bean bag toy	3.49
Bear Emporium	Bird bean bag toy	3.49
Bear Emporium	Rabbit bean bag toy	3.49
Bear Emporium	Raggedy Ann	4.99
Bear Emporium	King doll	9.49
Bear Emporium	Queen doll	9.49
Doll House Inc.	8 inch teddy bear	5.99
Doll House Inc.	12 inch teddy bear	8.99

Doll House Inc.	18 inch teddy bear	11.99
Doll House Inc.	Fish bean bag toy	3.49
Doll House Inc.	Bird bean bag toy	3.49
Doll House Inc.	Rabbit bean bag toy	3.49
Doll House Inc.	Raggedy Ann	4.99
Doll House Inc.	King doll	9.49
Doll House Inc.	Queen doll	9.49
Furball Inc.	8 inch teddy bear	5.99
Furball Inc.	12 inch teddy bear	8.99
Furball Inc.	18 inch teddy bear	11.99
Furball Inc.	Fish bean bag toy	3.49
Furball Inc.	Bird bean bag toy	3.49
Furball Inc.	Rabbit bean bag toy	3.49
Furball Inc.	Raggedy Ann	4.99
Furball Inc.	King doll	9.49
Furball Inc.	Queen doll	9.49
Fun and Games	8 inch teddy bear	5.99
Fun and Games	12 inch teddy bear	8.99
Fun and Games	18 inch teddy bear	11.99
Fun and Games	Fish bean bag toy	3.49
Fun and Games	Bird bean bag toy	3.49
Fun and Games	Rabbit bean bag toy	3.49
Fun and Games	Raggedy Ann	4.99
Fun and Games	King doll	9.49
Fun and Games	Queen doll	9.49
Jouets et ours	8 inch teddy bear	5.99
Jouets et ours	12 inch teddy bear	8.99
Jouets et ours	18 inch teddy bear	11.99
Jouets et ours	Fish bean bag toy	3.49
Jouets et ours	Bird bean bag toy	3.49
Jouets et ours	Rabbit bean bag toy	3.49
Jouets et ours	Raggedy Ann	4.99
Jouets et ours	King doll	9.49
Jouets et ours	Queen doll	9.49

## Analysis

As you can see in the preceding output, the Cartesian product is seldom what you want. The data returned here has matched every product with every vendor, including products with the incorrect vendor (and even vendors with no products at all).

### Caution: Don't Forget the WHERE Clause

Make sure all your joins have WHERE clauses, or the DBMS will return far more data than you want. Similarly, make sure your WHERE clauses are correct. An incorrect filter condition will cause the DBMS to return incorrect data.

### Tip: Cross Joins

Sometimes you'll hear the type of join that returns a Cartesian Product referred to as a cross join.

## Inner Joins

The join you have been using so far is called an equijoin—a join based on the testing of equality between two tables. This kind of join is also called an inner join. In fact, you may use a slightly different syntax for these joins, specifying the type of join explicitly. The following `SELECT` statement returns the exact same data as the preceding example:

### Input

[Click here to view code image](#)

```
SELECT vend_name, prod_name, prod_price
FROM Vendors INNER JOIN Products
ON Vendors.vend_id = Products.vend_id;
```

### Analysis

The `SELECT` in the statement is the same as the preceding `SELECT` statement, but the `FROM` clause is different. Here the relationship between the two tables is part of the `FROM` clause specified as `INNER JOIN`. When using this syntax the join condition is specified using the special `ON` clause instead of a `WHERE` clause. The actual condition passed to `ON` is the same as would be passed to `WHERE`.

Refer to your DBMS documentation to see which syntax is preferred.

### Note: The “Right” Syntax

Per the ANSI SQL specification, use of the `INNER JOIN` syntax is preferred over the simple equijoins syntax used previously. Indeed, SQL purists tend to look upon the simple syntax with disdain. That being said, DBMSs do indeed support both the simpler and the standard formats, so my recommendation is that you take the time to understand both formats, but use whichever you feel more comfortable with.

## Joining Multiple Tables

SQL imposes no limit to the number of tables that may be joined in a `SELECT` statement. The basic rules for creating the join remain the same. First list all the tables, and then define the relationship between each. Here is an example:

### Input

[Click here to view code image](#)

```
SELECT prod_name, vend_name, prod_price, quantity
FROM OrderItems, Products, Vendors
```

```
WHERE Products.vend_id = Vendors.vend_id
AND OrderItems.prod_id = Products.prod_id
AND order_num = 20007;
```

---

## Output

---

[Click here to view code image](#)

prod_name	vend_name	prod_price	quantity
-----	-----	-----	-----
18 inch teddy bear	Bears R Us	11.9900	50
Fish bean bag toy	Doll House Inc.	3.4900	100
Bird bean bag toy	Doll House Inc.	3.4900	100
Rabbit bean bag toy	Doll House Inc.	3.4900	100
Raggedy Ann	Doll House Inc.	4.9900	50

---

## Analysis

---

This example displays the items in order number 20007. Order items are stored in the OrderItems table. Each product is stored by its product ID, which refers to a product in the Products table. The products are linked to the appropriate vendor in the Vendors table by the vendor ID, which is stored with each product record. The FROM clause here lists the three tables, and the WHERE clause defines both of those join conditions. An additional WHERE condition is then used to filter just the items for order 20007.

### Caution: Performance Considerations

DBMSs process joins at run-time relating each table as specified. This process can become very resource intensive so be careful not to join tables unnecessarily. The more tables you join the more performance will degrade.

### Caution: Maximum Number of Tables in a Join

While it is true that SQL itself has no maximum number of tables per join restriction, many DBMSs do indeed have restrictions. Refer to your DBMS documentation to determine what restrictions there are, if any.

Now would be a good time to revisit the following example from [Lesson 11](#), “[Working with Subqueries](#).” As you will recall, this SELECT statement returns a list of customers who ordered product RGAN01:

---

## Input

---

[Click here to view code image](#)

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (SELECT cust_id
```

```
FROM Orders
WHERE order_num IN (SELECT order_num
                     FROM OrderItems
                     WHERE prod_id =
                        'RGAN01'));
```

Subqueries are not always the most efficient way to perform complex SELECT operations, and so as promised, here is the same query using joins:

## Input

[Click here to view code image](#)

```
SELECT cust_name, cust_contact
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
   AND OrderItems.order_num = Orders.order_num
   AND prod_id = 'RGAN01';
```

## Output

[Click here to view code image](#)

cust_name	cust_contact
Fun4All	Denise L. Stephens
The Toy Store	Kim Howard

## Analysis

As explained in [Lesson 11](#), returning the data needed in this query requires the use of three tables. But instead of using them within nested subqueries, here two joins are used to connect the tables. There are three WHERE clause conditions here. The first two connect the tables in the join, and the last one filters the data for product RGAN01.

### Tip: It Pays to Experiment

As you can see, there is often more than one way to perform any given SQL operation. And there is rarely a definitive right or wrong way. Performance can be affected by the type of operation, the DBMS being used, the amount of data in the tables, whether or not indexes and keys are present, and a whole slew of other criteria. Therefore, it is often worth experimenting with different selection mechanisms to find the one that works best for you.

## Summary

Joins are one of the most important and powerful features in SQL, and using them effectively requires a basic understanding of relational database design. In this lesson, you learned some of the basics of

relational database design as an introduction to learning about joins. You also learned how to create an equijoin (also known as an inner join), which is the most commonly used form of join. In the next lesson, you'll learn how to create other types of joins.

# Lesson 13. Creating Advanced Joins

*In this lesson, you'll learn all about additional join types—what they are, and how to use them. You'll also learn how to use table aliases and how to use aggregate functions with joined tables.*

## Using Different Join Types

So far, you have used only simple joins known as inner joins or *equijoins*. You'll Now take a look at three additional join types: the self join, the natural join, and the outer join.

### Self Joins

As I mentioned earlier, one of the primary reasons to use table aliases is to be able to refer to the same table more than once in a single `SELECT` statement. An example will demonstrate this. Suppose you wanted to send a mailing to all the customer contacts who work for the same company for which Jim Jones works. This query requires that you first find out which company Jim Jones works for, and next which customers work for that company. The following is one way to approach this problem:

#### Input

[Click here to view code image](#)

```
SELECT cust_id, cust_name, cust_contact
FROM Customers
WHERE cust_name = (SELECT cust_name
                   FROM Customers
                   WHERE cust_contact = 'Jim Jones');
```

#### Output

[Click here to view code image](#)

cust_id	cust_name	cust_contact
1000000003	Fun4All	Jim Jones
1000000004	Fun4All	Denise L. Stephens

#### Analysis

This first solution uses subqueries. The inner `SELECT` statement does a simple retrieval to return the `cust_name` of the company that Jim Jones works for. That name is the one used in the `WHERE` clause of the outer query so that all employees who work for that company are retrieved. (You learned all about subqueries in [Lesson 11](#), “[Working with Subqueries](#),” refer to that lesson for more information.)

Now look at the same query using a join:

#### Input

[Click here to view code image](#)

```
SELECT c1.cust_id, c1.cust_name, c1.cust_contact
FROM Customers AS c1, Customers AS c2
WHERE c1.cust_name = c2.cust_name
      AND c2.cust_contact = 'Jim Jones';
```

Output

[Click here to view code image](#)

cust_id	cust_name	cust_contact
-----	-----	-----
1000000003	Fun4All	Jim Jones
1000000004	Fun4All	Denise L. Stephens
,		

Analysis

The two tables needed in this query are actually the same table, and so the Customers table appears in the FROM clause twice. Although this is perfectly legal, any references to table Customers would be ambiguous because the DBMS does not know which Customers table you are referring to.

To resolve this problem table aliases are used. The first occurrence of Customers has an alias of C1, and the second has an alias of C2. Now those aliases can be used as table names. The SELECT statement, for example, uses the C1 prefix to explicitly state the full name of the desired columns. If it did not, the DBMS would return an error because there are two of each column named cust\_id, cust\_name, and cust\_contact. It cannot know which one you want. (Even though they are the same.) The WHERE clause first joins the tables and then filters the data by cust\_contact in the second table to return only the wanted data.

**Tip: Self-Joins Instead of Subqueries**

Self-joins are often used to replace statements using subqueries that retrieve data from the same table as the outer statement. Although the end result is the same, many DBMSs process joins far more quickly than they do subqueries. It is usually worth experimenting with both to determine which performs better.

Natural Joins

Whenever tables are joined, at least one column will appear in more than one table (the columns being used to create the join). Standard joins (the inner joins that you learned about in the last lesson) return all data, even multiple occurrences of the same column. A natural join simply eliminates those multiple occurrences so that only one of each column is returned.



How does it do this? The answer is it doesn't—you do it. A natural join is a join in which you select only columns that are unique. This is typically done using a wildcard (`SELECT *`) for one table and explicit subsets of the columns for all other tables. The following is an example:

## Input

---

[Click here to view code image](#)

```
SELECT C.*, O.order_num, O.order_date,  
       OI.prod_id, OI.quantity, OI.item_price  
FROM Customers AS C, Orders AS O,  
     OrderItems AS OI  
WHERE C.cust_id = O.cust_id  
     AND OI.order_num = O.order_num  
     AND prod_id = 'RGAN01';
```

---

### Tip: No AS in Oracle

Oracle users, remember to drop the AS.

---

## Analysis

---

In this example, a wildcard is used for the first table only. All other columns are explicitly listed so that no duplicate columns are retrieved.

The truth is, every inner join you have created thus far is actually a natural join, and you will probably never need an inner join that is not a natural join.

## Outer Joins

Most joins relate rows in one table with rows in another. But occasionally, you want to include rows that have no related rows. For example, you might use joins to accomplish the following tasks:

- Count how many orders were placed by each customer, including customers that have yet to place an order.
- List all products with order quantities, including products not ordered by anyone.
- Calculate average sale sizes, taking into account customers that have not yet placed an order.

In each of these examples, the join includes table rows that have no associated rows in the related table. This type of join is called an outer join.

### Caution: Syntax Differences

It is important to note that the syntax used to create an outer join can vary slightly among different SQL implementations. The various forms of syntax described in the following section cover most implementations, but refer to your DBMS documentation to verify its syntax before proceeding.

---

The following `SELECT` statement is a simple inner join. It retrieves a list of all customers and their

orders:

Input

---

[Click here to view code image](#)

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers INNER JOIN Orders
  ON Customers.cust_id = Orders.cust_id;
```

---

Outer join syntax is similar. To retrieve a list of all customers including those who have placed no orders, you can do the following:

Input

---

[Click here to view code image](#)

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers LEFT OUTER JOIN Orders
  ON Customers.cust_id = Orders.cust_id;
```

---

Output

---

[Click here to view code image](#)

cust_id	order_num
-----	-----
1000000001	20005
1000000001	20009
1000000002	NULL
1000000003	20006
1000000004	20007
1000000005	20008

Analysis

---

Like the inner join seen in the last lesson, this SELECT statement uses the keywords OUTER JOIN to specify the join type (instead of specifying it in the WHERE clause). But unlike inner joins, which relate rows in both tables, outer joins also include rows with no related rows. When using OUTER JOIN syntax you must use the RIGHT or LEFT keywords to specify the table from which to include all rows (RIGHT for the one on the right of OUTER JOIN, and LEFT for the one on the left). The previous example uses LEFT OUTER JOIN to select all the rows from the table on the left in the FROM clause (the Customers table). To select all the rows from the table on the right, you use a RIGHT OUTER JOIN as seen in this next example:

Input

---

[Click here to view code image](#)

```
SELECT Customers.cust_id, Orders.order_num
```

```
FROM Customers RIGHT OUTER JOIN Orders
ON Orders.cust_id = Customers.cust_id;
```

---

### Tip: Outer Join Types

Remember that there are always two basic forms of outer joins—the left outer join and the right outer join. The only difference between them is the order of the tables that they are relating. In other words, a left outer join can be turned into a right outer join simply by reversing the order of the tables in the `FROM` or `WHERE` clause. As such, the two types of outer join can be used interchangeably, and the decision about which one is used is based purely on convenience.

There is one other variant of the outer join, and that is the full outer join that retrieves all rows from both tables and relates those that can be related. Unlike a left outer join or right outer join, which includes unrelated rows from a single table, the full outer join includes unrelated rows from both tables. The syntax for a full outer join is as follows:

### Input

---

[Click here to view code image](#)

```
SELECT Customers.cust_id, Orders.order_num
FROM Orders FULL OUTER JOIN Customers
ON Orders.cust_id = Customers.cust_id;
```

---

### Caution: FULL OUTER JOIN Support

The `FULL OUTER JOIN` syntax is not supported by Access, MariaDB, MySQL, Open Office Base, or SQLite.

## Using Joins with Aggregate Functions

As you learned in [Lesson 9](#), “[Summarizing Data](#),” aggregate functions are used to summarize data. Although all the examples of aggregate functions thus far only summarized data from a single table, these functions can also be used with joins.

To demonstrate this, let’s look at an example. You want to retrieve a list of all customers and the number of orders that each has placed. The following code uses the `COUNT ( )` function to achieve this:

### Input

---

[Click here to view code image](#)

```
SELECT Customers.cust_id,  
       COUNT(Orders.order_num) AS num_ord  
FROM Customers INNER JOIN Orders  
  ON Customers.cust_id = Orders.cust_id  
GROUP BY Customers.cust_id;
```

---

### Output

---

[Click here to view code image](#)

cust_id	num_ord
-----	-----
10000000001	2
10000000003	1
10000000004	1
10000000005	1

---

### Analysis

---

This SELECT statement uses INNER JOIN to relate the Customers and Orders tables to each other. The GROUP BY clause groups the data by customer, and so the function call COUNT(Orders.order\_num) counts the number of orders for each customer and returns it as num\_ord.

Aggregate functions can be used just as easily with other join types. See the following example:

### Input

---

[Click here to view code image](#)

```
SELECT Customers.cust_id,  
       COUNT(Orders.order_num) AS num_ord  
FROM Customers LEFT OUTER JOIN Orders  
  ON Customers.cust_id = Orders.cust_id  
GROUP BY Customers.cust_id;
```

---

### Output

---

[Click here to view code image](#)

cust_id	num_ord
-----	-----
10000000001	2

1000000002	0
1000000003	1
1000000004	1
1000000005	1

## Analysis

---

This example uses a left outer join to include all customers, even those who have not placed any orders. The results show that customer 1000000002 is also included, this time with 0 orders.

## Using Joins and Join Conditions

Before I wrap up our two lesson discussion on joins, I think it is worthwhile to summarize some key points regarding joins and their use:

- Pay careful attention to the type of join being used. More often than not, you'll want an inner join, but there are often valid uses for outer joins, too.
- Check your DBMS documentation for the exact join syntax it supports. (Most DBMSs use one of the forms of syntax described in these two lessons.)
- Make sure you use the correct join condition (regardless of the syntax being used), or you'll return incorrect data.
- Make sure you always provide a join condition, or you'll end up with the Cartesian product.
- You may include multiple tables in a join and even have different join types for each. Although this is legal and often useful, make sure you test each join separately before testing them together. This will make troubleshooting far simpler.

## Summary

This lesson was a continuation of the last lesson on joins. This lesson started by teaching you how and why to use aliases, and then continued with a discussion on different join types and various forms of syntax used with each. You also learned how to use aggregate functions with joins, and some important do's and dont's to keep in mind when working with joins.

# Lesson 14. Combining Queries

*In this lesson, you'll learn how to use the UNION operator to combine multiple SELECT statements into one result set.*

## Understanding Combined Queries

Most SQL queries contain a single SELECT statement that returns data from one or more tables. SQL also enables you to perform multiple queries (multiple SELECT statements) and return the results as a single query result set. These combined queries are usually known as *unions* or *compound queries*. There are basically two scenarios in which you'd use combined queries:

- To return similarly structured data from different tables in a single query
- To perform multiple queries against a single table returning the data as one query

### Tip: Combining Queries and Multiple WHERE Conditions

For the most part, combining two queries to the same table accomplishes the same thing as a single query with multiple WHERE clause conditions. In other words, any SELECT statement with multiple WHERE clauses can also be specified as a combined query, as you'll see in the section that follows.

## Creating Combined Queries

SQL queries are combined using the UNION operator. Using UNION, multiple SELECT statements can be specified, and their results can be combined into a single result set.

### Using UNION

Using UNION is simple enough. All you do is specify each SELECT statement and place the keyword UNION between each.

Let's look at an example. You need a report on all your customers in Illinois, Indiana, and Michigan. You also want to include all Fun4All locations, regardless of state. Of course, you can create a WHERE clause that will do this, but this time you'll use a UNION instead.

As I just explained, creating a UNION involves writing multiple SELECT statements. First look at the individual statements:

### Input

[Click here to view code image](#)

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI');
```

### Output

[Click here to view code image](#)

cust_name	cust_contact	cust_email
-----	-----	-----
Village Toys	John Smith	sales@villagetoy.com
Fun4All	Jim Jones	jjones@fun4all.com
The Toy Store	Kim Howard	NULL

Input

[Click here to view code image](#)

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

Output

[Click here to view code image](#)

cust_name	cust_contact	cust_email
-----	-----	-----
Fun4All	Jim Jones	jjones@fun4all.com
Fun4All	Denise L. Stephens	dstephens@fun4all.com

Analysis

The first SELECT retrieves all rows in Illinois, Indiana, and Michigan by passing those state abbreviations to the IN clause. The second SELECT uses a simple equality test to find all Fun4All locations.

To combine these two statements, do the following:

Input

[Click here to view code image](#)

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI')
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

Output

[Click here to view code image](#)

cust_name	cust_contact	cust_email
-----	-----	-----

Fun4All	Denise L. Stephens	dstephens@fun4all.com
Fun4All	Jim Jones	jjones@fun4all.com
Village Toys	John Smith	sales@villagetoys.com
The Toy Store	Kim Howard	NULL

## Analysis

---

The preceding statements are made up of both of the previous `SELECT` statements separated by the `UNION` keyword. `UNION` instructs the DBMS to execute both `SELECT` statements and combine the output into a single query result set.

As a point of reference, here is the same query using multiple `WHERE` clauses instead of a `UNION`:

## Input

---

[Click here to view code image](#)

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI')
OR cust_name = 'Fun4All';
```

---

In our simple example, the `UNION` might actually be more complicated than using a `WHERE` clause. But with more complex filtering conditions, or if the data is being retrieved from multiple tables (and not just a single table), the `UNION` could have made the process much simpler indeed.

### Tip: UNION Limits

There is no standard SQL limit to the number of `SELECT` statements that can be combined with `UNION` statements. However, it is best to consult your DBMS documentation to ensure that it does not enforce any maximum statement restrictions of its own.

### Caution: Performance Issues

Most good DBMSs use an internal query optimizer to combine the `SELECT` statements before they are even processed. In theory, this means that from a performance perspective, there should be no real difference between using multiple `WHERE` clause conditions or a `UNION`. I say in theory, because, in practice, most query optimizers don't always do as good a job as they should. Your best bet is to test both methods to see which will work best for you.

## UNION Rules

As you can see, unions are very easy to use. But there are a few rules governing exactly which can be combined:

- A `UNION` must be composed of two or more `SELECT` statements, each separated by the keyword `UNION` (so, if combining four `SELECT` statements there would be three `UNION`



keywords used).

- Each query in a UNION must contain the same columns, expressions, or aggregate functions (and some DBMSs even require that columns be listed in the same order).
- Column datatypes must be compatible: They need not be the exact same type, but they must be of a type that the DBMS can implicitly convert (for example, different numeric types or different date types).

Aside from these basic rules and restrictions, unions can be used for any data retrieval tasks.

## Including or Eliminating Duplicate Rows

Go back to the preceding section titled “[Using UNION](#)” and look at the sample SELECT statements used. You’ll notice that when executed individually, the first SELECT statement returns three rows, and the second SELECT statement returns two rows. However, when the two SELECT statements are combined with a UNION, only four rows are returned, not five.

The UNION automatically removes any duplicate rows from the query result set (in other words, it behaves just as do multiple WHERE clause conditions in a single SELECT would). Because there is a Fun4All location in Indiana, that row was returned by both SELECT statements. When the UNION was used the duplicate row was eliminated.

This is the default behavior of UNION, but you can change this if you so desire. If you would, in fact, want all occurrences of all matches returned, you can use UNION ALL instead of UNION.

Look at the following example:

### Input

---

[Click here to view code image](#)

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI')
UNION ALL
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

---

### Output

---

[Click here to view code image](#)

cust_name	cust_contact	cust_email
-----	-----	-----
Village Toys	John Smith	sales@villagetoy.com
Fun4All	Jim Jones	jjones@fun4all.com
The Toy Store	Kim Howard	NULL
Fun4All	Jim Jones	jjones@fun4all.com
Fun4All	Denise L. Stephens	dstephens@fun4all.com

### Analysis

---

Using UNION ALL, the DBMS does not eliminate duplicates. Therefore, the preceding example returns five rows, one of them occurring twice.

**Tip: UNION versus WHERE**

AT the beginning of this lesson, I said that UNION almost always accomplishes the same thing as multiple WHERE conditions. UNION ALL is the form of UNION that accomplishes what cannot be done with WHERE clauses. If you do, in fact, want all occurrences of matches for every condition (including duplicates), you must use UNION ALL and not WHERE.

**Sorting Combined Query Results**

SELECT statement output is sorted using the ORDER BY clause. When combining queries with a UNION only one ORDER BY clause may be used, and it must occur after the final SELECT statement. There is very little point in sorting part of a result set one way and part another way, and so multiple ORDER BY clauses are not allowed.

The following example sorts the results returned by the previously used UNION:

**Input**

[Click here to view code image](#)

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI')
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All'
ORDER BY cust_name, cust_contact;
```

**Output**

[Click here to view code image](#)

cust_name	cust_contact	cust_email
-----	-----	-----
Fun4All	Denise L. Stephens	dstephens@fun4all.com
Fun4All	Jim Jones	jjones@fun4all.com
The Toy Store	Kim Howard	NULL
Village Toys	John Smith	sales@villagetoy.com

**Analysis**

This UNION takes a single ORDER BY clause after the final SELECT statement. Even though the ORDER BY appears to only be a part of that last SELECT statement, the DBMS will in fact use it to sort all the results returned by all the SELECT statements.

### **Note: Other UNION Types**

Some DBMSs support two additional types of UNION EXCEPT (sometimes called MINUS) can be used to only retrieve the rows that exist in the first table but not in the second, and INTERSECT can be used to retrieve only the rows that exist in both tables. In practice, however, these UNION types are rarely used as the same results can be accomplished using joins.

### **Tip: Working with Multiple Tables**

For simplicity's sake, the examples in this lesson have all used UNION to combine multiple queries on the same table. In practice, where UNION is really useful is when you need to combine data from multiple tables, even tables with mismatched column names, in which case you can combine UNION with aliases to retrieve a single set of results.

## **Summary**

In this lesson, you learned how to combine SELECT statements with the UNION operator. Using UNION, you can return the results of multiple queries as one combined query, either including or excluding duplicates. The use of UNION can greatly simplify complex WHERE clauses and retrieving data from multiple tables.