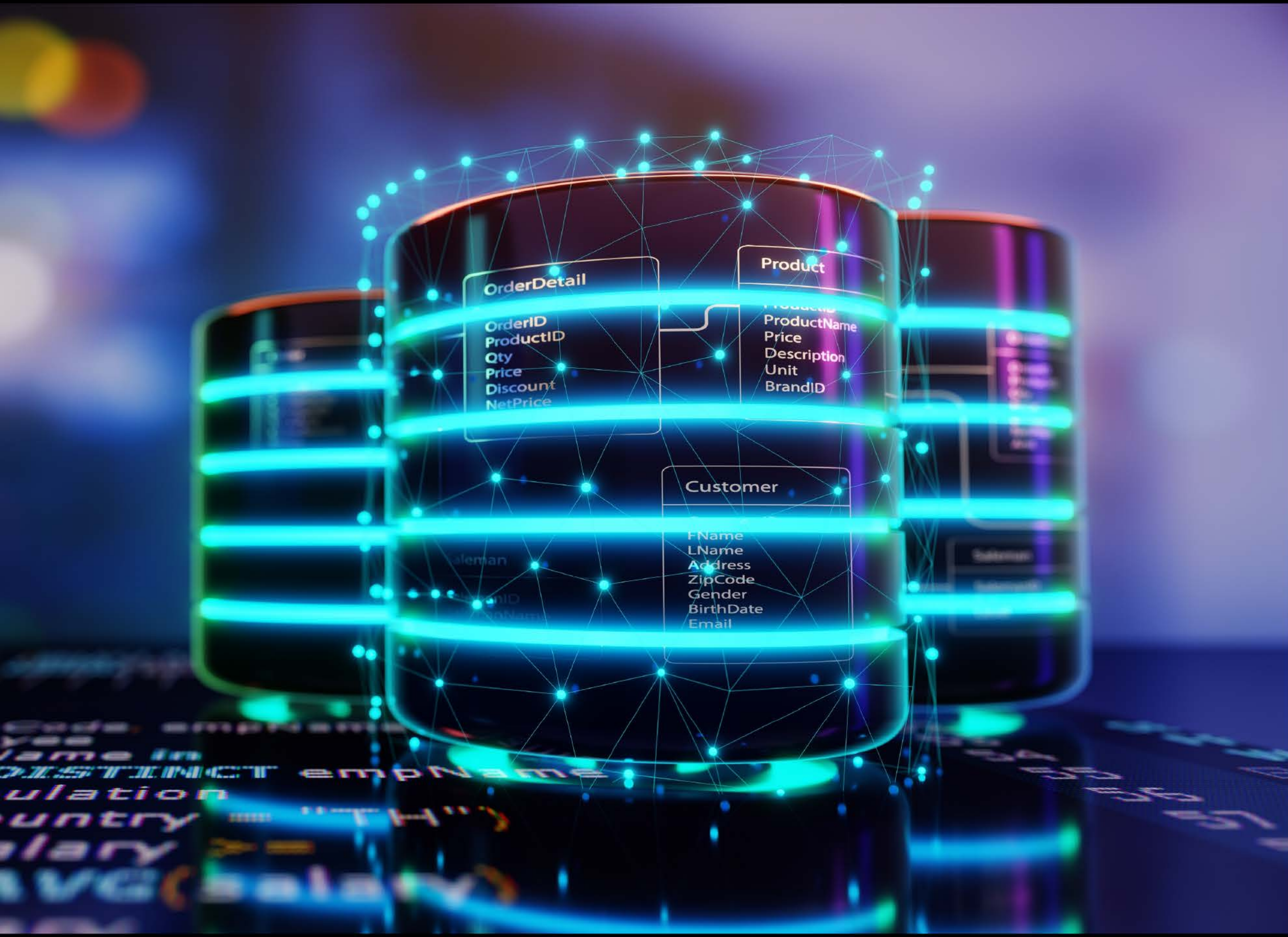
MySQL Essentials 9



Neil Smyth

Payload publishing

MySQL 9 Essentials

MySQL 9 Essentials

ISBN-13: 978-1-965764-10-7

Rev: 1.0

Payload
publishing

https://www.payloadbooks.com

Chapter 16

16. Understanding Joins and Unions

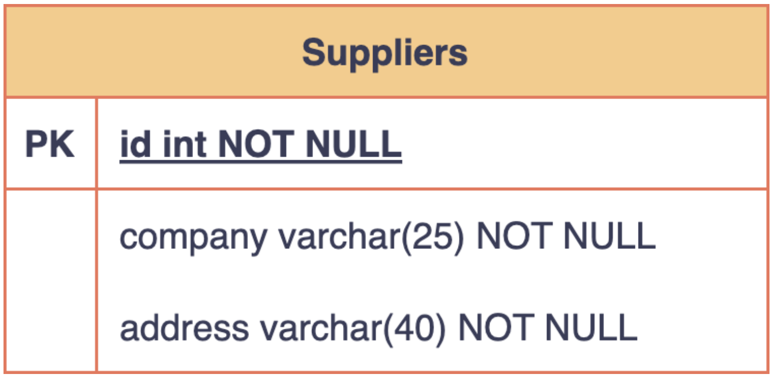
Up to this point, we have focused on creating a table that contains information about a single category—specifically, the products sold by an online electronics store. However, we would also need to store information about each product's supplier in a real-world scenario. Supplier data might include the supplier's name, address, and telephone number. We would also need to link these suppliers to the products they provide.

In this scenario, we have two options for storing contact information about the supplier. One option is to include the supplier's contact details in each row of the product table for every product sourced from that supplier. While this approach would be functional, it is highly inefficient, as it requires duplicating the supplier's contact information for every product they sell to us. Additionally, if the supplier relocates, we would have to update every row in the product table related to that supplier.

A more effective approach would be to create a separate supplier table that includes the contact information for each supplier. We can then reference this table when we want to retrieve supplier information for a specific product in the product table or list products by supplier. These connections between tables are known as a *joins* and are the foundation of relational databases.

16.1 How joins work

A join operates by leveraging the relationships between keys in different tables. For instance, in the above example, the supplier table includes a column called id, configured as the primary key, and the company name and address, as illustrated in Figure 16-1. (For more details on primary keys, see the chapter on "*The Basics of Databases*").



Suppliers	
PK	id int NOT NULL
	company varchar(25) NOT NULL address varchar(40) NOT NULL

Figure 16-1

The product table, however, stores information about our products, such as product ID, name, and description. Additionally, it contains a supplier_id column, which specifies the supplier from which each product is sourced. Since this supplier_id corresponds to a key in another table (the id key in the supplier table), it is known as a *foreign key*:

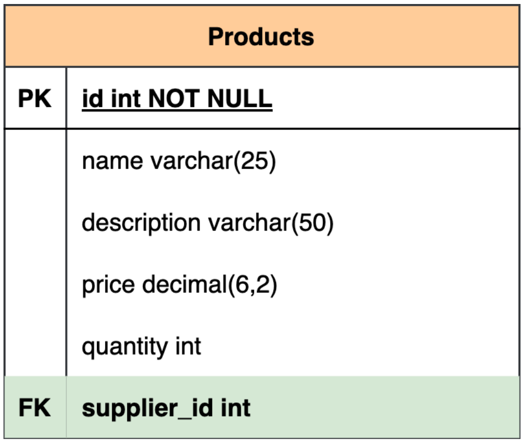


Figure 16-2

When writing a SELECT statement to retrieve data from the product table, we can use this foreign key to link it to the supplier table, extracting supplier information for each product. This approach allows us to combine data from both tables seamlessly, making it easier to analyze the relationships between products and their suppliers:

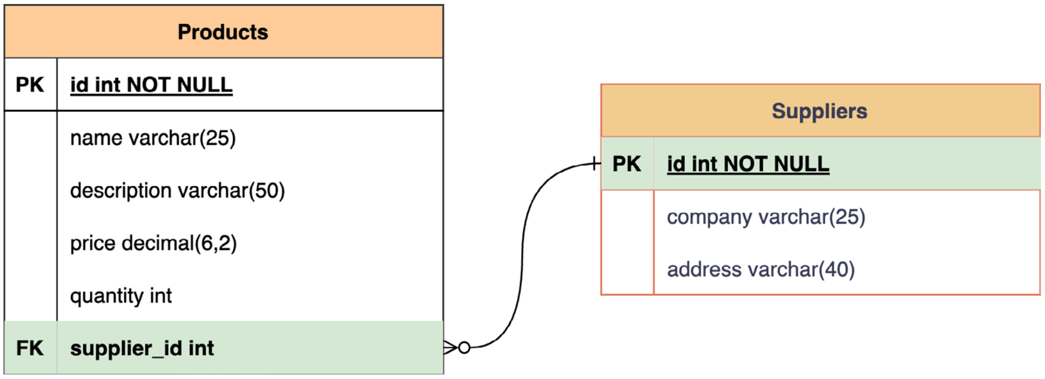


Figure 16-3

16.2 Opening the sample database

This chapter assumes that you have completed the steps in the previous chapters. If you haven't, you can import the current database snapshot using the sample files provided in the *"Start Here"* chapter.

To import the snapshot, open a terminal or command prompt, navigate to the directory containing the sample files, and run the following commands:

```
mysql -u root -p -e "CREATE DATABASE sampledb;"
mysql -u root -p sampledb < mysql_joins.sql
```

Once the database is ready, open the MySQL client and select the *sampledb* database:

```
USE sampledb
```

16.3 Creating the supplier table

Before exploring table joins in MySQL, we need to add a table to the *sampledb* database to contain supplier information. Using the MySQL client, execute the following statement to create the supplier table:

```
CREATE TABLE supplier
(
  id int NOT NULL AUTO_INCREMENT,
  company Varchar(25) NOT NULL,
  address VARCHAR(40) NULL,
  PRIMARY KEY (id)
);
```

After the supplier table has been created, use the following INSERT statement to add some supplier records:

```
INSERT INTO supplier VALUES (
   NULL, 'Apple', 'Cupertino, CA'
),
(
   NULL, 'Dell', 'Round Rock, TX'
),
(
   NULL, 'NZXT', 'City of Industry, CA'
),
(
   NULL, 'Corsair', 'Milpitas, CA'
);
```

16.4 Adding the foreign key to the product table

Our product table currently lacks a column to store the supplier ID reference, so use the following ALTER TABLE statement to add a column named supplier_id:

```
ALTER TABLE product ADD supplier_id INT;
```

Though we have added the supplier_id column, we still need to let MySQL know that it contains foreign keys. We do this using the FOREIGN KEY and REFERENCES keywords. These keywords enable us to define the new column as a foreign key and specify the corresponding table and column it refers to, thereby establishing a relationship between the product and supplier tables:

```
ALTER TABLE product
ADD FOREIGN KEY(supplier_id)
REFERENCES supplier(id);
```

16.5 Adding key values to supplier_id

The next step is to populate the supplier_id field of each product table row with the corresponding ID from the supplier table. Although the product row count is low enough that we could do this manually, the process would need to be automated for larger datasets. In this case, we use an UPDATE SET statement with the WHERE and LIKE operators to compare the first word of the product.description field (which consistently contains the manufacturer name) with the supplier.company column as follows:

```
UPDATE product
SET supplier_id = (SELECT id FROM supplier
WHERE product.description LIKE CONCAT(supplier.company, '%'));
```

In the statement above, we used the CONCAT() function to combine multiple strings into one string. In this case, the current company name is followed by the '%' wildcard. After the update, check that products with matching suppliers have been assigned the correct supplier id:

```
SELECT description, supplier_id FROM product;
```

```
+-------------------------+-------------+
| description             | supplier_id |
+-------------------------+-------------+
| Apple laptop            |           1 |
| Apple laptop            |           1 |
| Apple desktop           |           1 |
| Apple desktop           |           1 |
| Apple iPhone            |           1 |
| Apple iPhone            |           1 |
| Apple desktop           |           1 |
| NZXT PC case (gray)     |           3 |
| Corsair PC case (grey)  |           4 |
| Dell desktop            |           2 |
| Dell desktop            |           2 |
| Dell docking station    |           2 |
| One2One touchpad        |        NULL |
| One2One 2-port USB hub  |        NULL |
| One2One 4 port USB hub  |        NULL |
+-------------------------+-------------+
15 rows in set (0.00 sec)
```

With the table updates completed, we can begin exploring table joins.

16.6 Performing a cross join

When you join two tables using the CROSS JOIN clause, the result set will consist of all possible combinations of rows from both tables. Each row from the first table is paired with every row from the second table. The diagram in Figure 16-4 illustrates this concept:

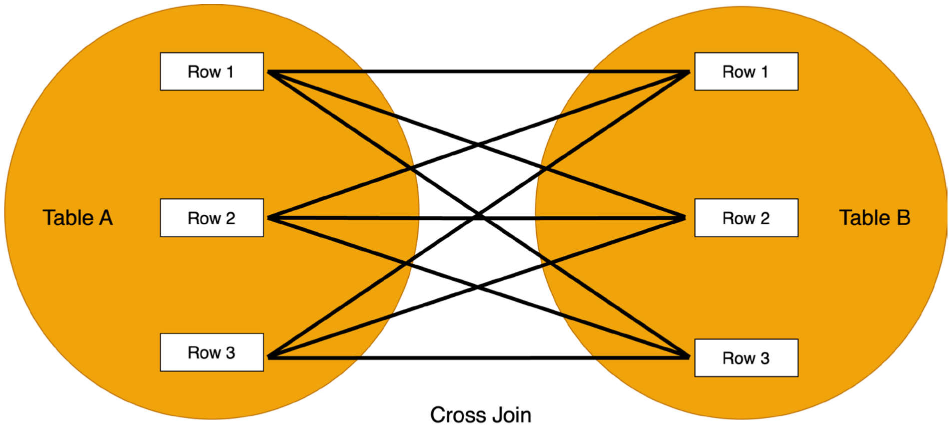


Figure 16-4

The SQL syntax for cross joins is as follows:

```
SELECT <column_names> FROM <table_1> CROSS JOIN <table_2>;
```

A cross join between our product and supplier tables will generate the following output (abbreviated for conciseness):

```
SELECT name, supplier.id from product CROSS JOIN supplier;
+-------------------------+----+
| name                    | id |
+-----------------------+----+
| MacBook Pro M4 14-in   |  4 |
| MacBook Pro M4 14-in   |  3 |
| MacBook Pro M4 14-in   |  2 |
| MacBook Pro M4 14-in   |  1 |
| MacBook Air M3         |  4 |
| MacBook Air M3         |  3 |
| MacBook Air M3         |  2 |
| MacBook Air M3         |  1 |
| Mac Mini M3            |  4 |
.
.
| One2One USB-A Hub      |  2 |
| One2One USB-A Hub      |  1 |
| One4One USB-A Hub      |  4 |
| One4One USB-A Hub      |  3 |
| One4One USB-A Hub      |  2 |
| One4One USB-A Hub      |  1 |
+-----------------------+----+
60 rows in set (0.01 sec)
```

Cross joins include all row combinations, including those for which tables are not matched. For example, the USB hub rows are listed in the above result set, even though no corresponding supplier exists for those products.

Note that we have to use what is known as the *fully qualified name* for the supplier id column since both tables contain an id column. A fully qualified column name is defined by specifying the table name followed by a dot (.) and then the column name (i.e., supplier.id).

16.7 Performing an inner join

The inner join combines rows from two or more tables based on comparisons between a specific column in each table, as illustrated in Figure 16-5:

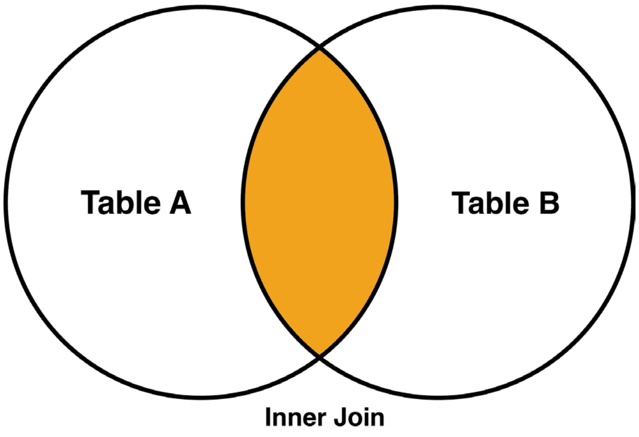


Figure 16-5

We could, for example, use an inner join to select rows from one or more tables that share a common customer phone number. Alternatively, a join could be established to retrieve rows where one table's price column value is less than another's. Inner joins use the following syntax:

```
SELECT <column_names>
 FROM <table_1>
```

```
   INNER JOIN <table_2>
    ON <column_1> <comparison_operator> <column_2>
   INNER JOIN <table_3>
    ON <column_3> <comparison_operator> <column_4> .. ;
```

The following statement, for example, retrieves the product name and supplier company from rows where the product.supplier_id and supplier.id columns match:

```
SELECT product.name, supplier.company
FROM product
INNER JOIN supplier
ON product.supplier_id = supplier.id;
+-------------------------+---------+
| name                    | company |
+-------------------------+---------+
| MacBook Pro M4 14-in    | Apple   |
| MacBook Air M3          | Apple   |
| Mac Mini M3             | Apple   |
| Mac Studio              | Apple   |
| iPhone 17 Pro           | Apple   |
| iPhone 17               | Apple   |
| iMac                    | Apple   |
| Dell XPS Model 4823     | Dell    |
| Dell XPS Model 7823     | Dell    |
| Dell Dock Model [4807]  | Dell    |
| NZXT Mid Tower          | NZXT    |
| Corsair Full Tower      | Corsair |
+-------------------------+---------+
12 rows in set (0.00 sec)
```

16.8 Performing left joins

A left join retrieves all records from the left table and the matching records from the right table. If no match is found in the right table, the corresponding columns in the result set will contain NULL values, guaranteeing that all rows from the left table are included, regardless of whether a match exists in the right table:

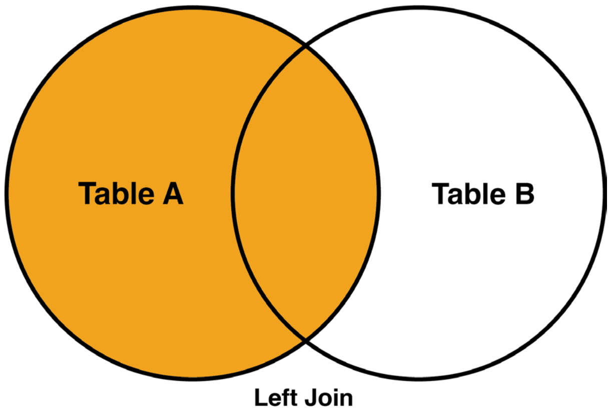


Figure 16-6

Left joins are created using the following syntax:

```
SELECT <column_names>
 FROM <table_1>
```

```
  LEFT JOIN <table_2>
   ON <column_1> <comparison_operator> <column_2>;
```

One key difference with LEFT JOIN is that it will include rows from the left table for which there is no match in the right table. For example, suppose we have rows in our product table for which there is no matching supplier in the supplier table. For example, when we run the following SELECT statement, the rows will still be displayed, but with NULL values for the supplier columns where no supplier exists:

```
SELECT product.name, supplier.company
FROM product
LEFT JOIN supplier
ON product.supplier_id = supplier.id;
+-------------------------+---------+
| name                    | company |
+-------------------------+---------+
| MacBook Pro M4 14-in    | Apple   |
| MacBook Air M3          | Apple   |
| Mac Mini M3             | Apple   |
| Mac Studio              | Apple   |
.
.
| Dell Dock Model [4807]  | Dell    |
| One&One Touch 200       | NULL    |
| One2One USB-A Hub       | NULL    |
| One4One USB-A Hub       | NULL    |
+-------------------------+---------+
15 rows in set (0.00 sec)
```

16.9 Performing right joins

As the name suggests, a right join achieves the opposite result of a left join. With a right join, all the rows from the second table (in this case, the supplier table) are included in the result set, regardless of any matching entries in the product table. If a supplier does not have any associated products in the product table, the corresponding columns from the product table will contain NULL values:

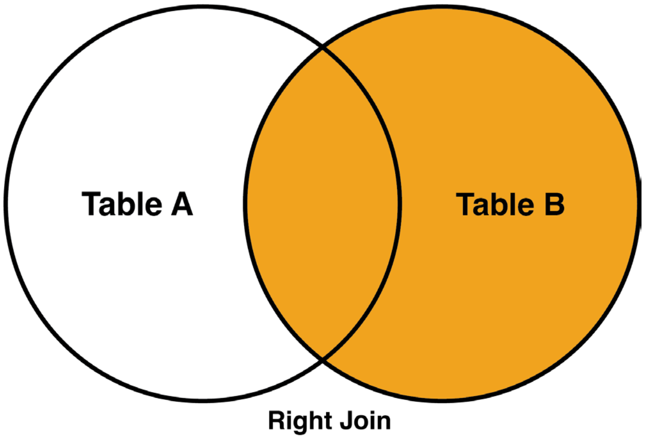


Figure 16-7

Right joins are formed using the following syntax:

```
SELECT <column_names>
 FROM <table_1>
```

```
  RIGHT JOIN <table_2>
   ON <column_1> <comparison_operator> <column_2>;
```

16.10 Understanding WHERE in join statements

The WHERE keyword allows us to narrow the results of table joins based on specific filtering criteria. Say, for example, that we want to list only products supplied by Dell. To do so, we would create a right join using the following statement:

```
SELECT product.name, supplier.company
FROM product
RIGHT JOIN supplier
ON product.supplier_id = supplier.id
WHERE company = 'Dell';
+-----------------------+---------+
| name                  | company |
+-----------------------+---------+
| Dell XPS Model 4823   | Dell    |
| Dell XPS Model 7823   | Dell    |
| Dell Dock Model [4807] | Dell    |
+-----------------------+---------+
3 rows in set (0.00 sec)
```

16.11 Working with unions

Although it is not considered a join, the UNION operator is a valuable tool for combining the results of multiple tables. As we will see later, it can also merge join statements. A UNION combines the result sets from two or more SELECT queries and follows this syntax:

```
SELECT <column_names> FROM <table_1>
 UNION
  SELECT <column_names> FROM <table_2>;
UNION
 SELECT ... ;
```

When using the UNION operator, there are several rules to follow:

- Each SELECT statement must return the same number of columns.
- The data types of the columns in the corresponding positions of each SELECT statement must be compatible. For example, you cannot mix INT and VARCHAR types in the same column position.
- The column names in each SELECT statement must be presented in the same order.

By default, the UNION operator will eliminate duplicate rows from the combined result set. If you want to keep duplicate rows, you can use the UNION ALL operator instead:

```
SELECT <column_names> FROM <table_1>
 UNION ALL
  SELECT <column_names> FROM <table_2> ... ;
```

To demonstrate unions, we will add a second supplier table to our sample database containing information about European suppliers. We will then create a union to retrieve the supplier addresses from both tables. This scenario is illustrated in Figure 16-8: