**Subqueries in MySQL**

A subquery is a type of query that is embedded—or nested—into a data manipulation language (DML) statement. The data returned by the subquery is passed into the DML statement and incorporated into its overall logic. The subquery itself is typically a SELECT statement, although you can also use a TABLE statement or VALUES statement. Even so, the SELECT statement continues to be the most common choice for subqueries, and that’s the one I focus on in this article.

**This article is part of Robert Sheldon's continuing series on Learning MySQL. To see all 11 items in the series,**[**click here**](https://www.red-gate.com/simple-talk/mysql-coding-basics/)**.**

With that in mind, let’s dive into the subquery and take a look at several different ones in action. In this article, I focus primarily on how the subquery is used in SELECT statements to retrieve data in different ways from one table or from multiple tables. Like the previous articles in this series, this one is meant to introduce you to the basic concepts of working with subqueries so you have a solid foundation on which to build your skills. Also like the previous articles, it includes a number of examples to help you better understand how to work with subqueries so you can start using them in your DML statements.

**Preparing your MySQL environment**

For the examples in this article, I used the same database and tables that I used for the previous article. The database is named travel and it includes two tables: manufacturers and airplanes. However, the sample data I use for this article is different from the last article, so I recommend that you once again rebuild the database and tables to keep things simple for this article’s examples. You can set up the database by running the following script:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | DROP DATABASE IF EXISTS travel;    CREATE DATABASE travel;  USE travel;  CREATE TABLE manufacturers (    manufacturer\_id INT UNSIGNED NOT NULL AUTO\_INCREMENT,    manufacturer VARCHAR(50) NOT NULL,    create\_date TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP,    last\_update TIMESTAMP NOT NULL      DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,    PRIMARY KEY (manufacturer\_id) )  ENGINE=InnoDB AUTO\_INCREMENT=1001;    CREATE TABLE airplanes (    plane\_id INT UNSIGNED NOT NULL AUTO\_INCREMENT,    plane VARCHAR(50) NOT NULL,    manufacturer\_id INT UNSIGNED NOT NULL,    engine\_type VARCHAR(50) NOT NULL,    engine\_count TINYINT NOT NULL,    max\_weight MEDIUMINT UNSIGNED NOT NULL,    wingspan DECIMAL(5,2) NOT NULL,    plane\_length DECIMAL(5,2) NOT NULL,    parking\_area INT GENERATED ALWAYS AS ((wingspan \* plane\_length)) STORED,    icao\_code CHAR(4) NOT NULL,    create\_date TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP,    last\_update TIMESTAMP NOT NULL      DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,    PRIMARY KEY (plane\_id),    CONSTRAINT fk\_manufacturer\_id FOREIGN KEY (manufacturer\_id)      REFERENCES manufacturers (manufacturer\_id) )  ENGINE=InnoDB AUTO\_INCREMENT=101; |

After you’ve created the database, you can add the sample data so you can follow along with the exercises in this article. Start by running the following INSERT statement to add data to the manufacturers table:

|  |  |
| --- | --- |
| 1  2  3 | INSERT INTO manufacturers (manufacturer)  VALUES ('Airbus'), ('Beagle Aircraft Limited'), ('Beechcraft'), ('Boeing'),    ('Bombardier'), ('Cessna'), ('Dassault Aviation'), ('Embraer'), ('Piper'); |

The statement adds nine rows to the manufacturers table, which you can confirm by querying the table. The manufacturer\_id value for the first row should be 1001. After you confirm the data in the manufacturers table, you can run the following INSERT statement to populate the airplanes table, using the manufacturer\_id values from the manufacturers table:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | INSERT INTO airplanes    (plane, manufacturer\_id, engine\_type, engine\_count,      wingspan, plane\_length, max\_weight, icao\_code)  VALUES    ('A340-600',1001,'Jet',4,208.17,247.24,837756,'A346'),    ('A350-800 XWB',1001,'Jet',2,212.42,198.58,546700,'A358'),    ('A350-900',1001,'Jet',2,212.42,219.16,617295,'A359'),    ('A380-800',1001,'Jet',4,261.65,238.62,1267658,'A388'),    ('A380-843F',1001,'Jet',4,261.65,238.62,1300000,'A38F'),    ('A.109 Airedale',1002,'Piston',1,36.33,26.33,2750,'AIRD'),    ('A.61 Terrier',1002,'Piston',1,36,23.25,2400,'AUS6'),    ('B.121 Pup',1002,'Piston',1,31,23.17,1600,'PUP'),    ('B.206',1002,'Piston',2,55,33.67,7500,'BASS'),    ('D.5-108 Husky',1002,'Piston',1,36,23.17,2400,'D5'),    ('Baron 56 TC Turbo Baron',1003,'Piston',2,37.83,28,5990,'BE56'),    ('Baron 58 (and current G58)',1003,'Piston',2,37.83,29.83,5500,'BE58'),    ('Beechjet 400 (same as MU-300-10 Diamond II)',1003,'Jet',2,43.5,48.42,15780,'BE40'),    ('Bonanza 33 (F33A)',1003,'Piston',1,33.5,26.67,3500,'BE33'),    ('Bonanza 35 (G35)',1003,'Piston',1,32.83,25.17,3125,'BE35'),    ('747-8F',1004,'Jet',4,224.42,250.17,987000,'B748'),    ('747-SP',1004,'Jet',4,195.67,184.75,696000,'B74S'),    ('757-300',1004,'Jet',2,124.83,178.58,270000,'B753'),    ('767-200',1004,'Jet',2,156.08,159.17,315000,'B762'),    ('767-200ER',1004,'Jet',2,156.08,159.17,395000,'B762'),    ('Learjet 24',1005,'Jet',2,35.58,43.25,13000,'LJ24'),    ('Learjet 24A',1005,'Jet',2,35.58,43.25,12499,'LJ24'),    ('Challenger (BD-100-1A10) 350',1005,'Jet',2,69,68.75,40600,'CL30'),    ('Challenger (CL-600-1A11) 600',1005,'Jet',2,64.33,68.42,36000,'CL60'),    ('Challenger (CL-600-2A12) 601',1005,'Jet',2,64.33,68.42,42100,'CL60'),    ('414A Chancellor',1006,'Piston',2,44.17,36.42,6750,'C414'),    ('421C Golden Eagle',1006,'Piston',2,44.17,36.42,7450,'C421'),    ('425 Corsair-Conquest I',1006,'Turboprop',2,44.17,35.83,8600,'C425'),    ('441 Conquest II',1006,'Turboprop',2,49.33,39,9850,'C441'),    ('Citation CJ1 (Model C525)',1006,'Jet',2,46.92,42.58,10600,'C525'),    ('EMB 175 LR',1008,'Jet',2,85.33,103.92,85517,'E170'),    ('EMB 175 Standard',1008,'Jet',2,85.33,103.92,82673,'E170'),    ('EMB 175-E2',1008,'Jet',2,101.67,106,98767,'E170'),    ('EMB 190 AR',1008,'Jet',2,94.25,118.92,114199,'E190'),    ('EMB 190 LR',1008,'Jet',2,94.25,118.92,110892,'E190'); |

The manufacturer\_id values from the manufacturers table provide the foreign key values needed for the manufacturer\_id column in the airplanes table. After you run the second INSERT statement, you can query the airplanes table to confirm that 35 rows have been added. The first row should have been assigned 101 for the plane\_id value, and the plane\_id values for the other rows should have been incremented accordingly.

**Building a basic scalar subquery**

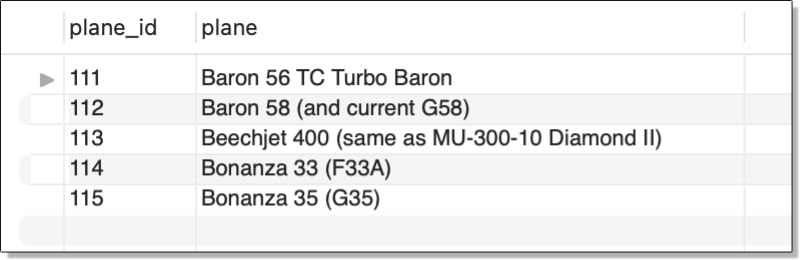
A scalar subquery is one that returns only a single value, which is then passed into the outer query through one of its clauses. The subquery is used in place of other possible expressions, such as a constants or column names. For example, the following SELECT statement (the outer query) includes a subquery in search condition of the WHERE clause:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT plane\_id, plane  FROM airplanes  WHERE manufacturer\_id =    (SELECT manufacturer\_id FROM manufacturers      WHERE manufacturer = 'Beechcraft'); |

The subquery is the expression on the right side of the equal sign, enclosed in parentheses. A subquery must always be enclosed in parentheses, no matter where it’s used in the outer statement.

Make certain that your subquery does indeed return only one value, if that’s what it’s supposed to do. If the subquery were to return multiple values and your WHERE clause is not set up to handle them (as in this example), MySQL will return an error letting you know that you messed up.

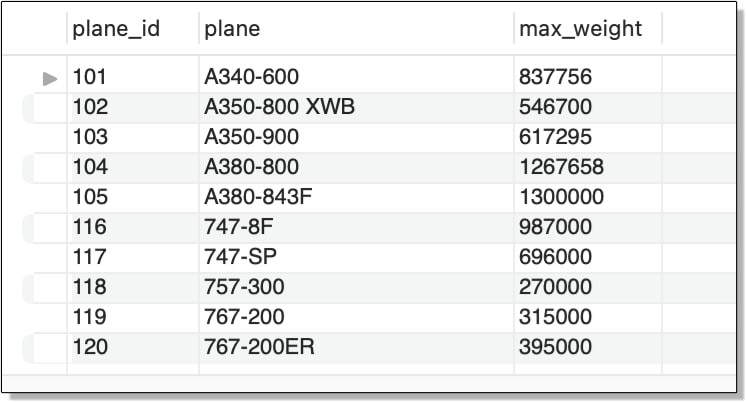
In this case, the subquery is a simple SELECT statement that returns the manufacturer\_id value for the manufacturer named Beechcraft. This value, 1003, is then passed into the WHERE clause as part of its search condition. If a row in the airplanes table contains a manufacturer\_id value that matches 1003, the row is included in the query results, which are shown in the following figure.



The subquery in this example retrieves data from a second table, in this case, manufacturers. However, a subquery can also retrieve data from the same table, which can be useful if the data must be handled in different ways. For example, the subquery in the following SELECT statement retrieves the average max\_weight value from the airplanes table:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT plane\_id, plane, max\_weight  FROM airplanes  WHERE max\_weight >    (SELECT AVG(max\_weight) FROM airplanes); |

The WHERE clause search condition in the outer statement uses the average to return only rows with a max\_weight value greater than that average. If you were to run the subquery on its own, you would see that the average maximum weight is 227,499 pounds. As a result, the outer SELECT statement returns only those rows with a max\_weight value that exceeds 227,499 pounds, as shown in the following figure.



As I mentioned earlier in the article, you can use subqueries in statements other than SELECT. One of those statements is the SET statement, which lets you assign variable values. You can use a SET statement to define a value that can then be passed into other statements. For example, the following SET and SELECT statements implement the same logic as the previous example and return the same results:

|  |  |
| --- | --- |
| 1  2  3  4 | SET @avg\_weight =    (SELECT ROUND(AVG(max\_weight)) FROM airplanes);  SELECT plane\_id, plane, max\_weight  FROM airplanes WHERE max\_weight > @avg\_weight; |

The SET statement defines a variable named @avg\_weight and uses a subquery to assign a value to that variable. This is the same subquery that is in the previous example. The SELECT statement then uses that variable in its WHERE clause (in place of the original subquery) to return only those rows with a max\_weight value greater than 227,499 pounds.

The examples in this section focused on using scalar subqueries in SELECT statements, but be aware that you can also use them in UPDATE and DELETE statements, as well as the SET clause of the UPDATE.

**Working with correlated subqueries**

One of the most valuable features of a subquery is its ability to reference the outer query from within the subquery. Referred to as a correlated subquery, this type of subquery can return data that is specific to the current row being evaluated by the outer statement. For example, the following SELECT statement uses a correlated subquery to calculate the average weight of the planes for each manufacturer, rather than for all planes:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT m.manufacturer\_id, m.manufacturer,    a.plane\_id, a.plane, a.max\_weight  FROM airplanes a INNER JOIN manufacturers m    ON a.manufacturer\_id = m.manufacturer\_id  WHERE a.max\_weight >    (SELECT ROUND(AVG(max\_weight)) FROM airplanes a2      WHERE a.manufacturer\_id = a2.manufacturer\_id); |

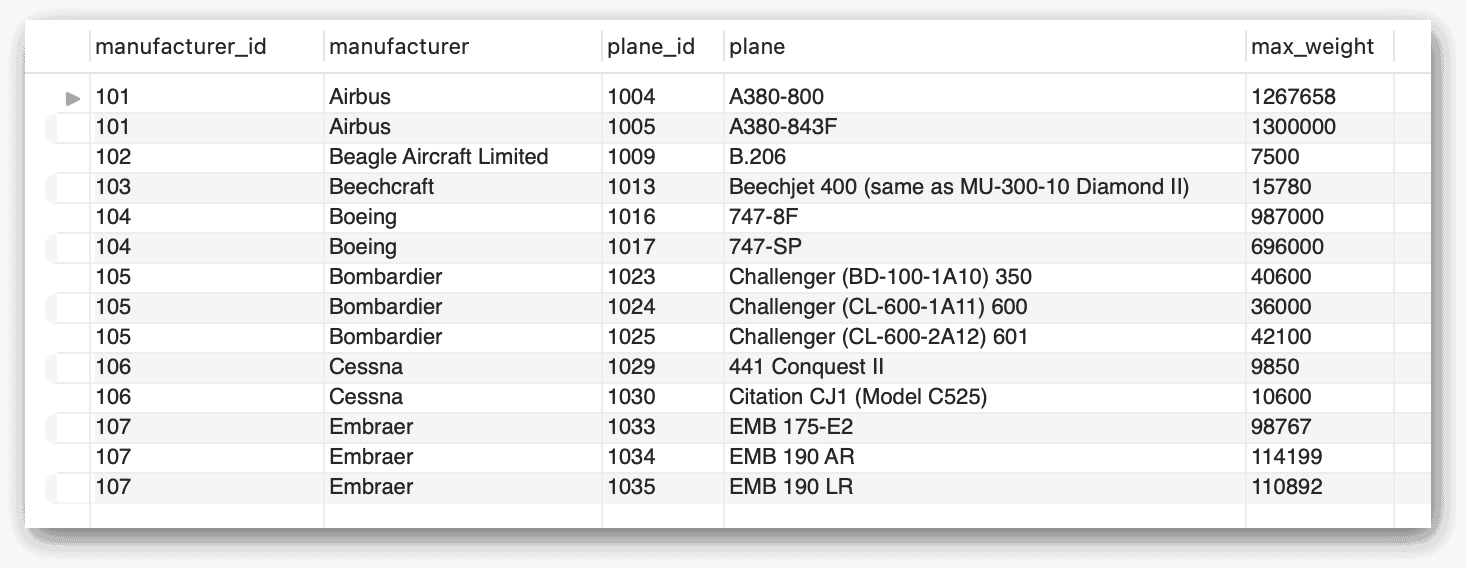
Unlike the previous two examples, the subquery now includes a WHERE clause that limits the returned rows to those with a manufacturer\_id value that matches the current manufacturer\_id value in the outer query. This is accomplished by assigning an alias (a) to the table in the outer query and using that alias when referencing the table’s manufacturer\_id column within the subquery.

In this case, I also assigned an alias (a2) to the table referenced within the subquery, but strictly speaking, you do not need to do this. I like to include an alias for consistency and code readability, but certainly take whatever approach works for you.

To better understand how the subquery works logically (as opposed to how the optimizer might actually execute it), consider the first row in the airplanes table, which has a manufacturer\_id value of 1001.

When the outer query evaluates the first row, it compares the max\_weight value to the value returned by the subquery. To carry out this comparison, the database engine first matches the manufacturer\_id value in the outer query to the manufacturer\_id values returned by the subquery’s SELECT statement. It then finds all rows associated with the current manufacturer and returns the average max\_weight value for that manufacturer, repeating the process for each manufacturer returned by the outer query.

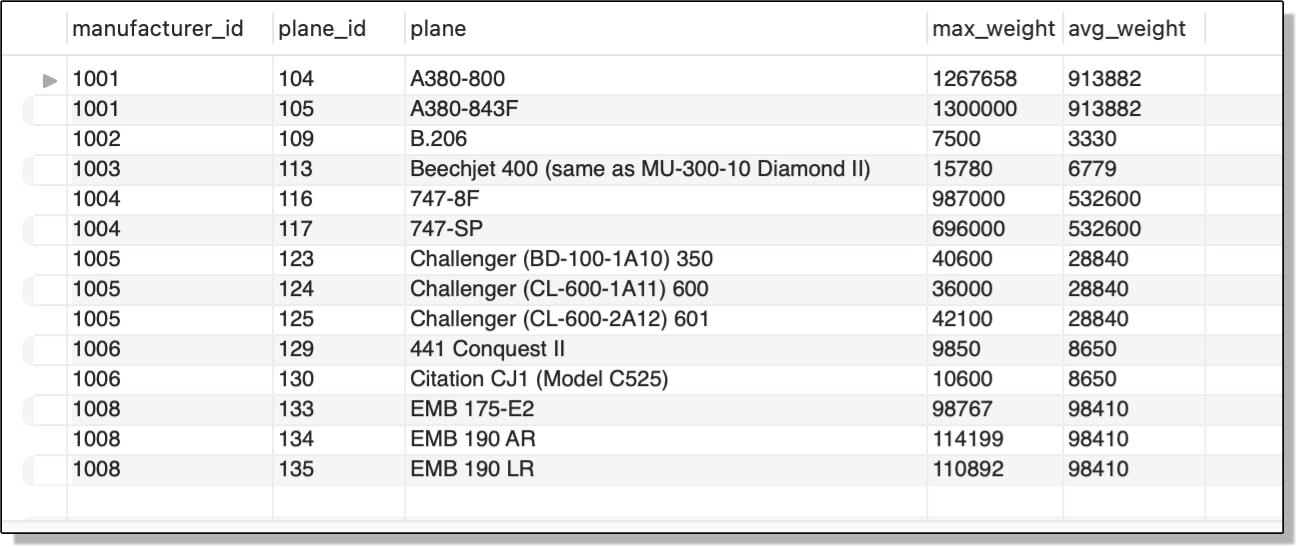
The following figure shows the results returned by the outer SELECT statement. The max\_weight values are now compared only with the manufacturer-specific averages.



You can also use this same subquery in the SELECT clause as one of the column expressions. In the following example, I added the subquery after the max\_weight column and assigned the alias avg\_weight to the new column:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT manufacturer\_id, plane\_id, plane, max\_weight,    (SELECT ROUND(AVG(max\_weight)) FROM airplanes a2      WHERE a.manufacturer\_id = a2.manufacturer\_id) AS avg\_weight  FROM airplanes a  WHERE max\_weight >    (SELECT ROUND(AVG(max\_weight)) FROM airplanes a2      WHERE a.manufacturer\_id = a2.manufacturer\_id); |

The new subquery works the same as in the preceding example. It returns the average weight only for the planes from the current manufacturer, as shown in the following figure.

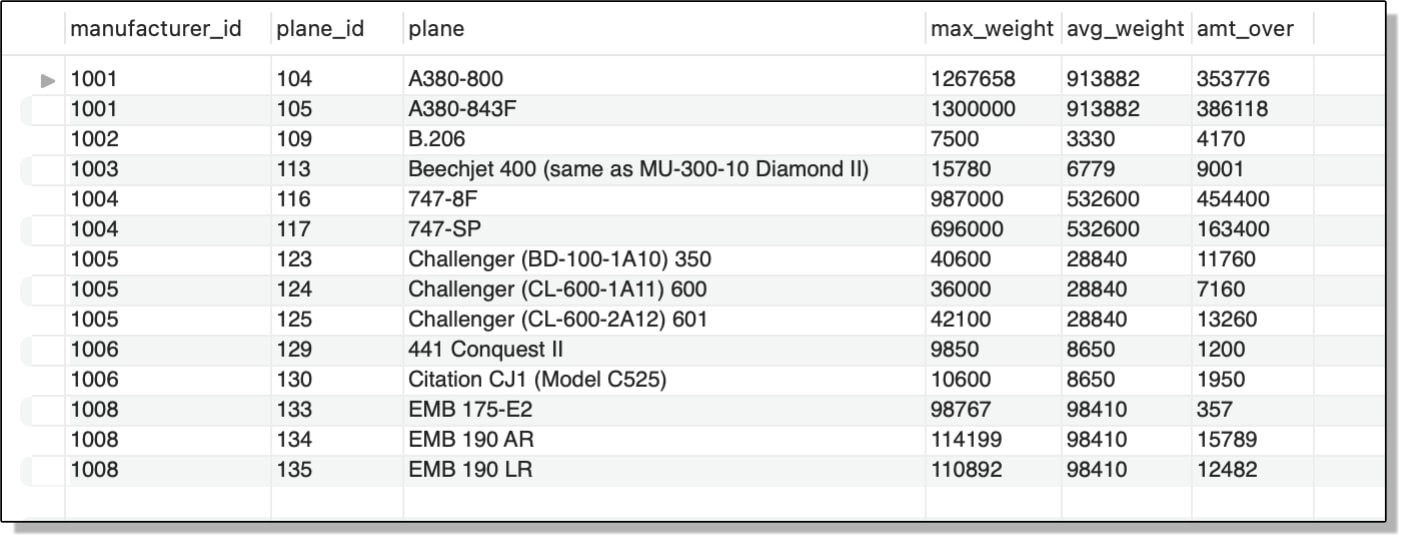


In the previous example, I used a subquery to create a generated column. However, you can use a subquery when creating an even more complex generated column. In the following example, I’ve added a generated column named amt\_over, which subtracts the average weight returned by the subquery from the weight in the max\_weight column:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | SELECT manufacturer\_id, plane\_id, plane, max\_weight,    (SELECT ROUND(AVG(max\_weight)) FROM airplanes a2      WHERE a.manufacturer\_id = a2.manufacturer\_id) AS avg\_weight,    -- subtracts average weight from max weight    (max\_weight - (SELECT ROUND(AVG(max\_weight)) FROM airplanes a2      WHERE a.manufacturer\_id = a2.manufacturer\_id)) AS amt\_over  FROM airplanes a  WHERE max\_weight >    (SELECT ROUND(AVG(max\_weight)) FROM airplanes a2      WHERE a.manufacturer\_id = a2.manufacturer\_id); |

The avg\_weight column in the SELECT list is a generated column that uses a subquery to return the average weight of the current manufacturer. The amt\_over column is also a generated column and it uses the same subquery to return the average weight. Only this time, the column subtracts that average from the max\_weight column to return the amount that exceeds the average.

The following figure shows the results now returned by the outer SELECT statement. As you can see, they include the amt\_over generated column, which shows the differences in the weights.



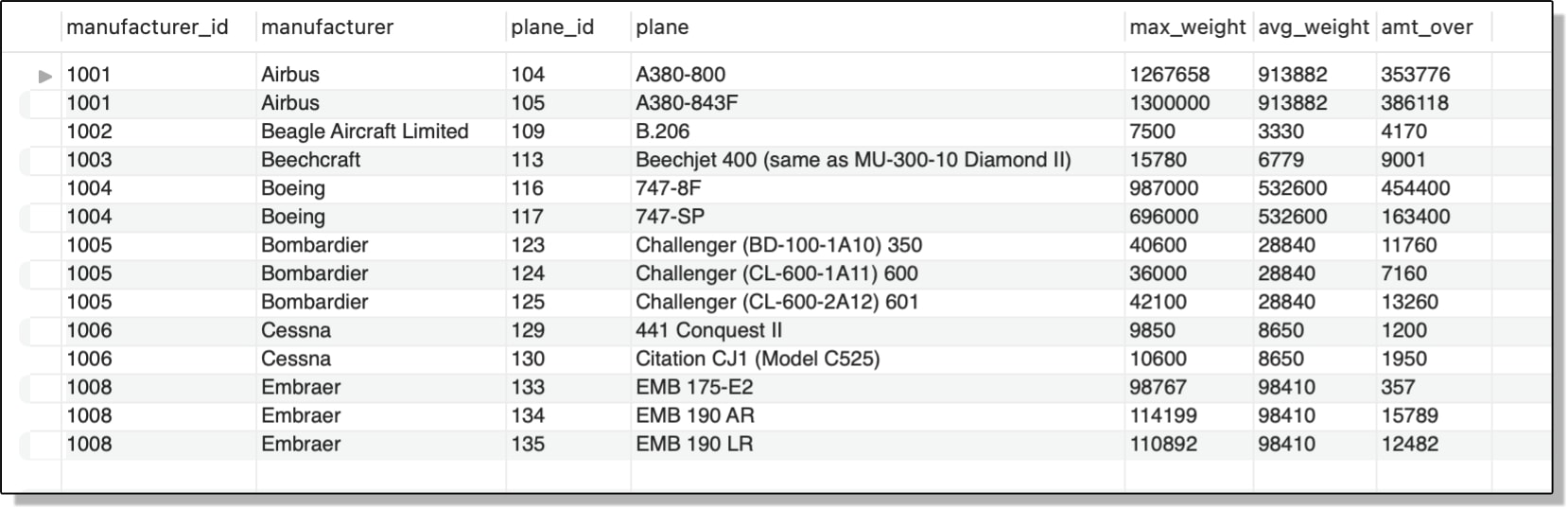
If you want, you can also retrieve the name of the manufacturer from the manufacturers table and include that in your SELECT clause, as shown in the following example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | SELECT manufacturer\_id,    (SELECT manufacturer FROM manufacturers m      WHERE a.manufacturer\_id = m.manufacturer\_id) manufacturer,    plane\_id, plane, max\_weight,    (SELECT ROUND(AVG(max\_weight)) FROM airplanes a2      WHERE a.manufacturer\_id = a2.manufacturer\_id) AS avg\_weight,    (max\_weight - (SELECT ROUND(AVG(max\_weight)) FROM airplanes a2      WHERE a.manufacturer\_id = a2.manufacturer\_id)) AS amt\_over  FROM airplanes a  WHERE max\_weight >    (SELECT ROUND(AVG(max\_weight)) FROM airplanes a2      WHERE a.manufacturer\_id = a2.manufacturer\_id); |

This statement is similar to the previous statement except that it adds the manufacturer column, a generated column. The new column uses a subquery to retrieve the name of the manufacturer from the manufacturers table, based on the manufacturer\_id value returned by the outer query.

In some cases, a subquery might not perform as well as other types of constructs. For example, MySQL can often optimize a left outer join better than a subquery that carries out comparable logic. If you’re using a subquery to perform an operation that can be achieved in another way and are concerned about performance, you should consider testing both options under a realistic workload to determine which is the best approach.

The following figure shows the results with the additional column, which I’ve named manufacturer.



One other detail I want to point out about using subqueries in the SELECT list is that you can include them even if you’re grouping and aggregating data. For instance, the following SELECT statement includes a subquery that retrieves the name of the manufacturer associated with each group:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | SELECT manufacturer\_id,    (SELECT manufacturer FROM manufacturers m      WHERE a.manufacturer\_id = m.manufacturer\_id) manufacturer,    COUNT(\*) AS plane\_amt  FROM airplanes a  WHERE engine\_type = 'piston'  GROUP BY manufacturer\_id  ORDER BY plane\_amt DESC; |

The subquery in this statement is similar to those you’ve seen in other examples, except that now you’re dealing with aggregated data, so the subquery must use a column that is mentioned in the GROUP BY clause of the outer statement, which it does (the manufacturer\_id column). The statement returns the results shown in the following figure.



As you can see, the data has been grouped based on the manufacturer\_id column, and the name of the manufacturer is included with each ID. In addition, the number of airplanes with piston engines is provided for each manufacturer, with the results sorted in descending order, based on that amount.

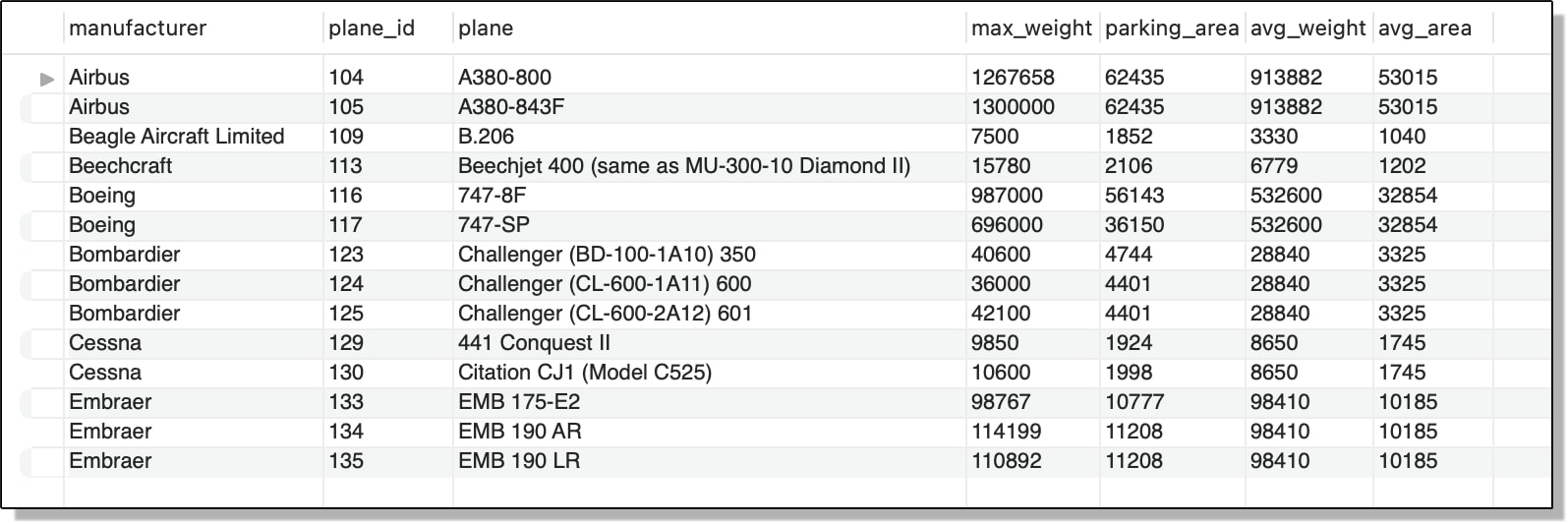
**Working with a row of data**

So far, all the examples in this article have returned scalar values, but your subqueries can also return multiple values, as noted earlier. For example, it might be useful to use a subquery to aggregate a table’s data, calculate specific averages in that data (returned as a single row), and then retrieve rows from the same table that exceed those averages, which is what I’ve done in the following SELECT statement:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | SELECT    (SELECT manufacturer FROM manufacturers m      WHERE a.manufacturer\_id = m.manufacturer\_id) manufacturer,  plane\_id, plane, max\_weight, parking\_area,    (SELECT ROUND(AVG(max\_weight)) FROM airplanes a2      WHERE a.manufacturer\_id = a2.manufacturer\_id) AS avg\_weight,    (SELECT ROUND(AVG(parking\_area)) FROM airplanes a2      WHERE a.manufacturer\_id = a2.manufacturer\_id) AS avg\_area  FROM airplanes a  WHERE (max\_weight, parking\_area) >    (SELECT ROUND(AVG(max\_weight)), ROUND(AVG(parking\_area))      FROM airplanes a2      WHERE a.manufacturer\_id = a2.manufacturer\_id); |

Notice that the WHERE clause in the outer statement includes the max\_weight and parking\_area columns in parentheses. In MySQL, this is how you create what is called a *row constructor,* a structure that supports simultaneous comparisons of multiple values. In this case, the row constructor is compared to the results from the subquery, which returns two corresponding values. The first value is the average weight, as you saw earlier. The second value returns the average parking area, which is based on the values in the parking\_area column.

In both cases, the averages are specific to the current manufacturer\_id value in the outer query. For the outer query to return a row, the two values in the row constructor must be greater than both corresponding values returned by the subquery. Notice also that the SELECT list now includes both the max\_weight and parking\_area columns, along with the average for each one. The outer statement returns the results shown in the following figure.



One thing you might have noticed about the subqueries in the preceding examples, particularly those used in the WHERE clauses, is that all the search conditions in those clauses use basic comparison operators, either equal (=) or greater than (>). However, you can use any of the other comparison operators, including special operators such as IN and EXISTS, as you’ll see in the next section.

**Working with a column of data**

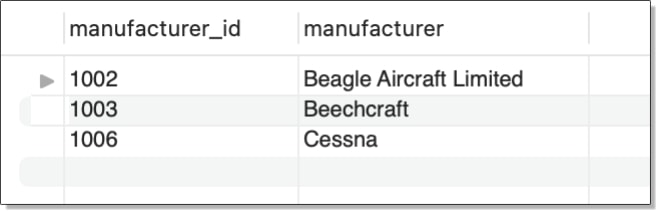
The previous section covered row subqueries. A row subquery can return only a single row, although that row can include one or more columns. This is in contrast to a scalar subquery , which returns only a single row and single column. In this section, we’ll look at the column subquery, which returns only a one column with one or more rows.

As with row subqueries, column subqueries are often used in the WHERE clause when building your search conditions. Also like row subqueries, your search condition must take into account that the subquery is returning more than one value.

For example, the WHERE clause in the following SELECT statement uses the IN operator to compare current the manufacturer\_id value in the outer statement with the list of manufacturer\_id values returned by the subquery:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT manufacturer\_id, manufacturer  FROM manufacturers  WHERE manufacturer\_id IN    (SELECT DISTINCT manufacturer\_id FROM airplanes      WHERE engine\_type = 'piston'); |

The column data returned by the subquery includes only those manufacturers that offer planes with piston engines (as reflected in the current data set). The IN operator determines whether the current manufacturer\_id value is included in that list. If it is, that row is returned, as shown in the following figure.



As with many operations in MySQL, you can take different approaches to achieve the same results. For example, you can replace the IN operator with an equal comparison operator, followed by the ANY keyword, as shown in the following example:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT manufacturer\_id, manufacturer  FROM manufacturers  WHERE manufacturer\_id = ANY    (SELECT DISTINCT manufacturer\_id FROM airplanes      WHERE engine\_type = 'piston'); |

You could have instead used another comparison operator, such as greater than (>) or lesser than (<) or even ALL instead of ANY. The statement returns the same results as the previous example. In fact, you can also achieve the same results by rewriting the entire statement as an inner join:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT DISTINCT m.manufacturer\_id, m.manufacturer  FROM manufacturers m INNER JOIN airplanes a    ON m.manufacturer\_id = a.manufacturer\_id  WHERE a.engine\_type = 'piston'; |

I’m not going to go into joins here, but as I mentioned earlier, joins can sometimes provide performance benefits over subqueries, so you should be familiar with how they work and how they differ from subqueries (a topic that could easily warrant its own article).

With that in mind, be aware that you can also use the NOT keyword with some operators to return different results. For example, the WHERE clause in the following SELECT statement uses the NOT IN operator ensure the current manufacturer\_id value is not in the list of values returned by the subquery:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT manufacturer\_id, manufacturer  FROM manufacturers  WHERE manufacturer\_id NOT IN    (SELECT DISTINCT manufacturer\_id FROM airplanes); |

In this case, the subquery returns a distinct list of all manufacturer\_id values in the airplanes table. The list is then compared to each manufacturer\_id value in the manufacturers table, as specified by the outer query. If the value is not in the list, the search condition evaluates to true and the row is returned. In this way, you can determine which manufacturers are in the manufacturers table but are not in the airplanes table. The query results are shown in the following figure.

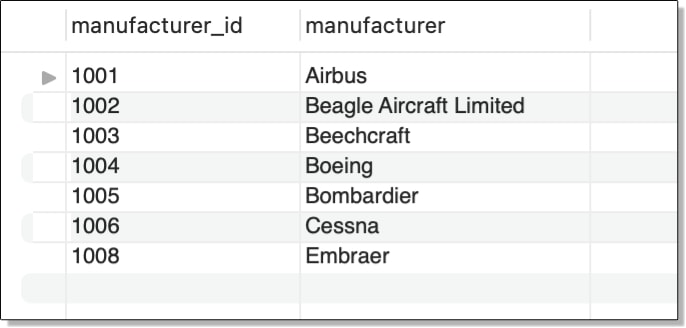


Be very careful when using NOT IN with your subqueries. If the subquery returns a NULL value, your WHERE expression will never evaluate to true.

Another operator you can use in the WHERE clause is EXISTS (and its counterpart NOT EXISTS). The EXISTS operator simply checks whether the subquery returns any rows. If it does, the search condition evaluates to true, otherwise it evaluates to false. For example, the following SELECT statement defines similar logic as the preceding example, except that it checks for which manufacturers are included in both tables:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT manufacturer\_id, manufacturer  FROM manufacturers m  WHERE EXISTS    (SELECT \* FROM airplanes A      WHERE a.manufacturer\_id = m.manufacturer\_id); |

Notice that you need only specify the EXISTS operator followed by the subquery. If the subquery returns a row for the current manufacturer\_id value, the search condition evaluates true and the outer query returns a row for that manufacturer. The following figure shows the results returned by the statement.



When working with column subqueries, it’s important to understand how to use comparison operators such as IN, NOT IN, ANY, ALL, and EXISTS. If you’re not familiar with them, be sure to refer to the MySQL documentation to learn more.

**Using subqueries in the FROM clause**

In addition to rows, columns, and scalar values, subqueries can also return tables, which are referred to as derived tables. Like other subqueries, a table subquery must be enclosed in parentheses. In addition, it must also be assigned an alias, similar to specifying a table alias when building correlated subqueries. For example, the following SELECT statement includes a table subquery named total\_planes, which is included in the FROM clause of the outer statement:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT ROUND(AVG(amount), 2) avg\_amt  FROM    (SELECT manufacturer\_id, COUNT(\*) AS amount    FROM airplanes    GROUP BY manufacturer\_id) AS total\_planes; |

Notice that the outer statement does not specify a table other than the derived table returned by the subquery. The subquery itself groups the data in the airplanes table by the manufacturer\_id values and then returns the ID and total number of planes in each group. The outer statement then finds the average number of planes across all groups. In this case, the statement returns a value of 5.00.

Now let’s look at another example of a table subquery. Although this next example is similar to the previous one in several ways, it includes something you have not seen yet, one subquery nested within another. In this case, I’ve nested a table subquery within another table subquery to group data based on custom categories and then find the average across those groups:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | -- outer SELECT calculates average count across all categories  SELECT    ROUND(AVG(amount), 2) AS avg\_amt  FROM    -- outer subquery aggregates categories and calculates count for each one    (SELECT category, COUNT(\*) amount    FROM      -- inner subquery categorizes planes based on parking area      (SELECT CASE        WHEN parking\_area > 50000 THEN 'A'        WHEN parking\_area >= 20000 THEN 'B'        WHEN parking\_area >= 10000 THEN 'C'        WHEN parking\_area >= 5000 THEN 'D'        WHEN parking\_area >= 1000 THEN 'E'        ELSE 'F'      END AS category      FROM airplanes) AS plane\_size    GROUP BY category    ORDER BY category) AS plane\_cnt; |

The innermost subquery—the one with the CASE expression—assigns one of five category values (A, B, C, D, E, and F) to each range of parking area values. The subquery returns a derived table named plane\_size, which contains a single column named category. The column contains a category value for each plane in the airplanes table.

The data from the plane\_size table is then passed to the outer subquery. This subquery groups the plane\_size data based on the category values and generates a second column named amount, which provides the total number of planes in each category. The outer subquery returns a derived table named plane\_cnt. The outer statement then finds the average number of planes across all groups in the derived table, returning a value of 5.83.