**Introduction**

In the last chapter, we saw how the various types of SQL joins can "connect" multiple tables in a single SELECT query. This powerful technique gives us fine‑grained control over the data our SQL queries return. However, SQL offers much more. In this chapter, we'll use the PACKT ONLINE SHOP database to see how

* subqueries
* case statements
* views

boost the flexibility and power of SQL, in an efficient, intuitive way. As we proceed, we'll have two activities

* Activity One
* Activity Two

and various exercises.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Subqueries**

In a SQL statement, a subquery operates like a joined table. It gathers data in a structured way, and makes that data available for its outer SQL query. However, a subquery offers flexibility of its own, because it is not restricted to an existing database table. Although queries with joins often have better performance, a subquery structure can seem a little more intuitive at first glance. Additionally, it's always good to have another option available when we build our queries. Both MySQL and SQL Server handle subqueries. Although a SQL subquery is a complete, stand-alone SQL statement of its own, we'll focus on subqueries that return only one column. Placed in parentheses, a subquery, or "inner" query, nests inside a parent, or "outer" query. A subquery can nest in the

* SELECT
* FROM
* WHERE

clause of its parent query, and it can nest in another subquery. SQL Server supports up to 32 subquery nesting levels. In SQL, all SELECT, UPDATE, INSERT, and DELETE statements support subqueries. A subquery can certainly return two or more columns, but if the outer query expects only one column, the subquery must return only one column.

We'll use a subquery to solve a real-world problem using the PACKT\_ONLINE\_SHOP database, comparing that solution with a join-based solution we'll see first. Suppose a manager wants a list of products that have zero sales. As we start in, the PRODUCTS and ORDERITEMS tables probably have the information we need, because ORDERITEMS ties specific products to specific orders. Using what we learned about SQL joins, we can build a query to answer the question. The query

USE packt\_online\_shop;

SELECT Products.ProductID, Products.ProductName,

Products.ProductCategoryID

FROM Products LEFT OUTER JOIN

OrderItems

ON Products.ProductID = OrderItems.ProductID

WHERE OrderItems.ProductID IS NULL

ORDER BY Products.ProductID;

works for both MySQL 8.0.15 and SQL Server 2014. In this query, we placed the PRODUCTS table columns we want in the select clause. The left outer join matches the rows between the PRODUCTS and ORDERITEMS tables. It also combines those rows with PRODUCTS table rows that don't match with ORDERITEMS table rows. The query left outer joined the tables on their ProductID columns. In the where clause, IS NULL filtered out the PRODUCTS table rows that have a matching row in the OrderItems table. This makes sense, because we want the PRODUCTS table rows that don't appear in the ORDERITEMS table. The order by clause sorts the result set

**ProductID ProductName, ProductCategoryID**

'4' 'The Gourmet Crockpot' '1'

'8' 'tomato sauce' '3'

'9' 'pure vanilla' '1'

by ProductID. We can also solve the problem with a subquery.

This query

USE packt\_online\_shop;

SELECT Products.ProductID, Products.ProductName,

Products.ProductCategoryID

FROM Products

WHERE Products.ProductID NOT IN

(SELECT ProductID FROM OrderItems)

ORDER BY Products.ProductID;

also works for both MySQL 8.0.15 and SQL Server 2014. We'll focus on the inner query - the subquery - first, and then work out from there. In this query, we started with a complete SQL query

SELECT ProductID FROM OrderItems

that shows all the ProductID values in the OrderItems table. We placed this query inside parentheses to make it a subquery. In the outer query, the select and from clauses have the PRODUCTS table columns we want. The outer query where clause uses "not in" to filter out PRODUCTS table rows that don't exist in the subquery. The order by clause sorted the result set by ProductID.

**Exercise 1: Use a Subquery to Find the Customers Who Have Not Placed an Order**

Note

This exercise is designed for MySQL 8.0.15, but the concepts and syntax will work for SQL Server 2014.

1. We'll start with the ORDERS table, since the ORDERS table ties the orders together with the customers table. Build a list of CustomerID values for all customers in the ORDERS table. Test it, and put it aside for now:

USE packt\_online\_shop;

SELECT CustomerID FROM ORDERS;

2. Build a first name / last name list of all customers in the CUSTOMERS table. In the CONCAT function, add a space to format the name. Although we won't really need it, include the CustomerID as a good practice:

USE packt\_online\_shop;

SELECT CONCAT(customers.FirstName, ' ',

customers.LastName) AS 'name',

customers.CustomerID

FROM customers

3. Add a where clause to the query, using "not in" to filter the CUSTOMERS table with the subquery that we built first. Sort the result set by name:

USE packt\_online\_shop;

SELECT CONCAT(customers.FirstName, ' ',

customers.LastName) AS 'name',

customers.CustomerID

FROM customers

WHERE customers.customerID NOT IN

(SELECT CustomerID FROM ORDERS)

ORDER BY 'name';

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Case Statements**

As we build SQL queries and statements, often enough we'll need a way to examine a set of conditions, and then return a value based on one of those conditions. The MySQL and SQL Server CASE statement does exactly this. A SQL CASE statement will step through a list of conditions, and then return a result based on the first matching condition it finds. Case statements work in SELECT, UPDATE, DELETE, WHERE, and HAVING clauses, and they operate a lot like simple IF-ELSE statements. Note the CASE statement can't control execution flow in a SQL function or stored procedure. We'll see more about functions and stored procedures later on.

We can get "basic" product information from the products table, but now we need a list of products that labels them by NetRetailPrice. Specifically, we need to use these

if NetRetailPrice is less than or equal to 24.99, then label it 'Cheap'

if NetRetailPrice is more than 24.99 and less than or equal to 79.99, then label it 'Mid-price'

if NetRetailPrice is more than 79.99 and less than or equal to 2499.99, then label it 'Expensive'

if NetRetailPrice has any other price, then label it 'Very Expensive'

rules. This

SELECT ProductName, WholesalePrice, NetRetailPrice,

'Price Point' AS 'Price Point', UnitKGWeight

FROM products

ORDER BY ProductName;

SQL query shows the

* ProductName
* WholesalePrice
* NetRetailPrice
* UnitKGWeight

columns from the products table, plus a new column called 'Price Point' that will hold the pricing label we want. This query will certainly work as is, but that 'Price Point' column will literally show 'Price Point' for every row. The SQL CASE statement will help. If we add the CASE statement to the query

SELECT ProductName, WholesalePrice, NetRetailPrice,

CASE

WHEN NetRetailPrice <= 24.99 THEN 'Cheap'

WHEN NetRetailPrice > 24.99 AND

NetRetailPrice <= 79.99 THEN 'Mid-price'

WHEN NetRetailPrice > 79.99 AND

NetRetailPrice <= 2499.99 THEN 'Expensive'

ELSE 'Very Expensive'

END AS 'Price Point',

UnitKGWeight

FROM products

ORDER BY ProductName;

we'll solve the problem. The CASE statement starts with the keyword "CASE" and ends with an "END". It has one or more conditions, one for each rule that we want to test. Each condition starts with the "WHEN" keyword, then the condition to test, followed by the "THEN" keyword, and finally the result for that specific condition. As soon as the CASE statement finds a true condition, it will execute that condition and leave the statement. The "ELSE" keyword serves as a default if no conditions are true. In SQL Server, a CASE statement column does not require a column name alias, but we should always include it. MySQL requires a CASE statement column name alias.

**Exercise 2: Use a CASE statement to label products by their shipping cost**

Note

This exercise is designed for MySQL 8.0.15, but the concepts and syntax will work for SQL Server 2014.

The Packt Online Shop calculates its product shipping prices based on this

NetRetailPrice \* UnitKGWeight

calculation. In a SELECT statement, we can place this in a CASE statement column to label each product by its shipping cost with these

if (NetRetailPrice \* UnitKGWeight) is less than or equal to 1.0, then label it 'Cheap'

if (NetRetailPrice \* UnitKGWeight) is more than 1.0 and

less than or equal to 35.00, then label it 'Mid-price'

if (NetRetailPrice \* UnitKGWeight) is more than 35.00 and less than or equal to 100.00, then label it 'Expensive'

if (NetRetailPrice \* UnitKGWeight) has any other price, then label it 'Very Expensive'

rules.

1. We'll start with the PRODUCTS table. First, build a basic list of products, their net retail price, and unit kilogram values:

USE packt\_online\_shop;

SELECT ProductName, NetRetailPrice, UnitKGWeight

FROM products;

2. Add a 'Shipping Cost' column to the SELECT statement. Add only one condition to test the overall idea, and remember that we'll need a calculation. Don't worry about NULL values in the 'Shipping Cost' column:

USE packt\_online\_shop;

SELECT ProductName, NetRetailPrice, UnitKGWeight,

CASE

WHEN (NetRetailPrice \* UnitKGWeight) <= 1.0 THEN 'Cheap'

END AS 'Shipping Cost'

FROM products

3. Everything looks good, so add in all the conditions. Remember the "ELSE" condition to cover the default:

USE packt\_online\_shop;

SELECT ProductName, NetRetailPrice, UnitKGWeight,

CASE

WHEN (NetRetailPrice \* UnitKGWeight) <= 1.0 THEN 'Cheap'

WHEN (NetRetailPrice \* UnitKGWeight) > 1.0 AND

(NetRetailPrice \* UnitKGWeight) <= 35.00 THEN 'Mid-price'

WHEN (NetRetailPrice \* UnitKGWeight) > 35.00 AND (NetRetailPrice \* UnitKGWeight) <= 100.00 THEN 'Expensive'

ELSE 'Very Expensive'

END AS 'Shipping Cost'

FROM products

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Views**

SQL views can encapsulate complex queries, exposing the columns in a much cleaner way for use by other queries. Views can also limit access to the tables "behind" them, which increases security. Additionally, a SQL view can make database maintenance easier if different queries and resources throughout the database use that view. One change in one view could substitute changes in potentially hundreds of other database queries.

As a business grows, its management financial management and reporting needs will probably grow as well. The management will turn to the database for answers. Every database query will be different, but eventually, folks will realize that the same, or very similar, SELECT statements seem to come up again and again. The same SELECT statement could show up in a subquery, or maybe in a stored procedure, something we'll learn about soon. That statement could become really complicated, with a lot of tables, outer joins, CASE statements of its own, and more. If we could somehow save that SELECT statement, and then use it as like a table as a basis for other SQL statements and queries, we could save a lot of time and effort. SQL views will help solve the problem.

This query

USE packt\_online\_shop;

SELECT CONCAT(customers.FirstName, ' ', customers.LastName)

AS 'CustomerName',

orders.OrderDate, products.ProductName

FROM customers INNER JOIN orders ON

customers.CustomerID = orders.CustomerID

INNER JOIN orderitems ON

orders.OrderID = orderitems.OrderID

INNER JOIN products ON

orderitems.ProductID = products.ProductID

will work for both MySQL 8.0.15 and SQL Server 2014, and it could appear again and again. To help, MySQL and SQL Server offer views. Think of a SQL view as a predefined SELECT statement with one or more tables, and at least one column from one table. A view returns only table columns. Once defined, other SQL queries and statements can use it as another table. A SQL view can isolate tables and columns, securely exposing only those data resources that other stakeholders might need. Additionally, if multiple database resources use the same view, and the tables behind the view change, we would only need to change the view itself, in a one-to-many way. Nothing that relies on the view would have to change. Without a view, we would change each of those resources, which would become a major pain point.

**Exercise 3: Build a SQL view that shows Packt Online Shop customer and their per-product spending**

Note

Although this exercise has almost identical syntax for both MySQL 8.0.15 and SQL Server 2014, we'll see an extra line for the SQL Server version that won't work for the MySQL version.

The Packt Online Shop now has a lot of queries, reports, etc. that rely on the same list of customers, their order dates, and the amount they spent on each product in each order. Although not required, we can see that if we include ProductID and ProductName information in the list, it will help us in the future. We'll build a view to handle this.

1. First, build and test a basic SQL query for the list we need:

USE PACKT\_ONLINE\_SHOP;

SELECT customers.CustomerID, orders.OrderDate,

products.ProductID, products.ProductName,

orderitems.Quantity \* orderitems.UnitPrice AS

'PerProductSpending'

FROM customers INNER JOIN orders ON

customers.CustomerID = orders.CustomerID

INNER JOIN orderitems ON

orders.OrderID = orderitems.OrderID

INNER JOIN products ON

orderitems.ProductID = products.ProductID

2. Add statements to build a view around this tested SQL statement. Name it "PACKT\_VIEW\_1" and use the

CREATE VIEW {***view name***} AS

syntax. For MySQL, it will look like this:

USE PACKT\_ONLINE\_SHOP;

CREATE VIEW PACKT\_VIEW\_1

AS

SELECT customers.CustomerID, orders.OrderDate,

products.ProductID, products.ProductName,

orderitems.Quantity \* orderitems.UnitPrice AS

'PerProductSpending'

FROM customers INNER JOIN orders ON

customers.CustomerID = orders.CustomerID

INNER JOIN orderitems ON

orders.OrderID = orderitems.OrderID

INNER JOIN products ON

orderitems.ProductID = products.ProductID

3. Now build the SQL Server version. Before SQL Server executes the CREATE VIEW statement, the keyword "GO" after the

USE PACKT\_ONLINE\_SHOP;

command tells SQL Server to "point" to the PACKT\_ONLINE\_SHOP database as a separate batch of SQL statements. In SQL Server, the CREATE VIEW statement requires the GO command. Don't use it for the MySQL version.

In this sample, everything after the "--" is a comment, which will not execute. We'll see more about this later:

USE PACKT\_ONLINE\_SHOP;

GO -- Include for SQL Server; don't include for MySQL

CREATE VIEW PACKT\_VIEW\_1

AS

SELECT customers.CustomerID, orders.OrderDate,

products.ProductID, products.ProductName,

orderitems.Quantity \* orderitems.UnitPrice AS

'PerProductSpending'

FROM customers INNER JOIN orders ON

customers.CustomerID = orders.CustomerID

INNER JOIN orderitems ON

orders.OrderID = orderitems.OrderID

INNER JOIN products ON

orderitems.ProductID = products.ProductID

4. For the MySQL version from the MySQL version, remove the "GO" command after the

USE PACKT\_ONLINE\_SHOP;

command:

USE PACKT\_ONLINE\_SHOP;

CREATE VIEW PACKT\_VIEW\_1

AS

SELECT customers.CustomerID, orders.OrderDate,

products.ProductID, products.ProductName,

orderitems.Quantity \* orderitems.UnitPrice AS

'PerProductSpending'

FROM customers INNER JOIN orders ON

customers.CustomerID = orders.CustomerID

INNER JOIN orderitems ON

orders.OrderID = orderitems.OrderID

INNER JOIN products ON

orderitems.ProductID = products.ProductID

4. Test the PACKT\_VIEW\_1 view:

USE PACKT\_ONLINE\_SHOP;

SELECT CustomerID, OrderDate, ProductID, ProductName,

PerProductSpending

FROM PACKT\_VIEW\_1

WHERE PerProductSpending > 14.99