Practice with Intermediate SQL

Chapter 7

Michael Tsiang

Stats 167: Introduction to Databases

UCLA

Intermediate SQL Exercises

Conditional Statements

Intermediate SQL Exercises

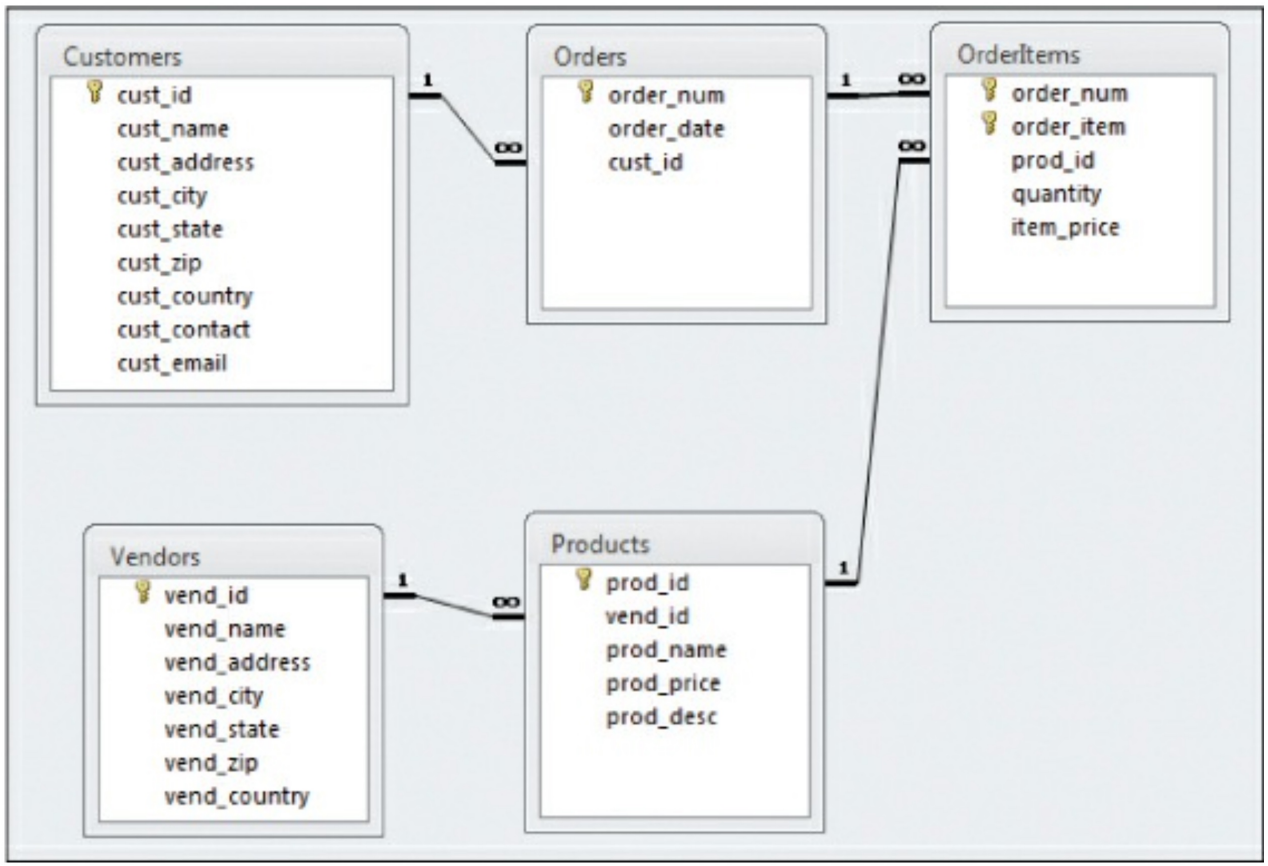
Brave New World

In the previous chapter, we unlocked a powerful set of tools in SQL. By learning subqueries, joins, and compound queries, we can now leverage the relationships in relational databases to answer more complex questions beyond simple queries.

To build proficiency in these new tools, we can practice our understanding with some sample exercises.

Schema of the TYSQL Data

We will continue to use the TYSQL database, which contains data for a fictitious toy store. The schema of the TYSQL database:



Source: Ben Forta, *SQL in 10 Minutes a Day, Sams Teach Yourself, 5th Edition*, Pearson, 2020.

Exercise 1: Bougie Customers

Suppose we are considering sending our wealthier customers targeted advertisements for our more expensive products.

Using a subquery, return a list of customers (by customer ID) who bought items priced at $10 or more.

Exercise 1: Bougie Customers (Solution)

```
SELECT cust_id
FROM Orders
WHERE order_num IN (SELECT order_num
                    FROM OrderItems
                    WHERE item_price >= 10);
```

cust_id
1000000001
1000000003
1000000004
1000000005

Exercise 2: Total Recall

Suppose there is a recall on a certain batch of the product with ID `BR01`. We need to know the dates when product `BR01` was ordered.

Write a SQL statement that uses a subquery to return the customer ID and order date for each order in which `BR01` was purchased, and sort the results by order date.

Exercise 2: Total Recall (Solution)

```sql
SELECT cust_id, order_date
FROM Orders
WHERE order_num IN (SELECT order_num
                    FROM OrderItems
                    WHERE prod_id = 'BR01')
ORDER BY order_date;
```

cust_id	order_date
1000000003	2020-01-12
1000000001	2020-05-01

Follow-up Question: If we need to notify the customers of the recall, how can we update this query to return the customer email for any customers who purchased items with a product ID `BR01`?

Exercise 2: Total Recall (Follow-Up Solution 1)

```sql
SELECT cust_email
FROM Customers
WHERE cust_id IN (SELECT cust_id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num
                                      FROM OrderItems
                                      WHERE prod_id = 'BR01'));
```

cust_email
sales@villagetoys.com
jjones@fun4all.com

Exercise 3: Biggest Supporters

We appreciate customers who have supported our business the most, so we need a list of customer IDs with the total amount they have ordered.

Write a SQL statement to return customer ID and the total amount they have ordered for each customer, sorting the results by amount spent from greatest to the least.

Exercise 3: Biggest Supporters (Solution 1)

```
SELECT cust_id,
       SUM(item_price * quantity) AS total_ordered
FROM Orders, OrderItems
WHERE Orders.order_num = OrderItems.order_num
GROUP BY cust_id
ORDER BY total_ordered DESC;
```

cust_id	total_ordered
1000000001	3515.5
1000000004	1696.0
1000000003	329.6
1000000005	189.6

Follow-up Question: How can we equivalently write this query as an explicit join?

Exercise 3: Biggest Supporters (Solution 2)

As an explicit inner join:

```
SELECT cust_id,
       SUM(item_price * quantity) AS total_ordered
FROM Orders
INNER JOIN OrderItems
ON Orders.order_num = OrderItems.order_num
GROUP BY cust_id
ORDER BY total_ordered DESC;
```

cust_id	total_ordered
1000000001	3515.5
1000000004	1696.0
1000000003	329.6
1000000005	189.6

Exercise 3: Biggest Supporters (Solution 3)

Since `order_num` is the same name in both `Orders` and `OrderItems` tables, we can also use a natural join:

```
SELECT cust_id,
       SUM(item_price * quantity) AS total_ordered
FROM Orders
NATURAL JOIN OrderItems
GROUP BY cust_id
ORDER BY total_ordered DESC;
```

cust_id	total_ordered
1000000001	3515.5
1000000004	1696.0
1000000003	329.6
1000000005	189.6

Exercise 4: Subquery to Join

In an earlier problem, we found a statement using subqueries to return the customer email for any customers who purchased items with a product ID `BR01`:

```
SELECT cust_email
FROM Customers
WHERE cust_id IN (SELECT cust_id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num
                                      FROM OrderItems
                                      WHERE prod_id = 'BR01'));
```

cust_email
sales@villagetoys.com
jjones@fun4all.com

How can we equivalently write this query as a join?

Exercise 4: Subquery to Join (Solution)

```sql
SELECT cust_email
FROM Customers
INNER JOIN Orders
ON Customers.cust_id = Orders.cust_id
INNER JOIN OrderItems
ON Orders.order_num = OrderItems.order_num
WHERE prod_id = 'BR01';
```

cust_email
sales@villagetoys.com
jjones@fun4all.com

Exercise 5: Vendor Audit

Suppose we want to assess the relationships we have with our vendors. We need to know how many products we sell from each vendor.

Write a SQL statement to list the vendors (by name) and the number of products they have available, including vendors with no products.

Exercise 5: Vendor Audit (Solution)

```sql
SELECT vend_name, COUNT(prod_id) AS num_prods
FROM Vendors
LEFT OUTER JOIN Products
ON Vendors.vend_id = Products.vend_id
GROUP BY Vendors.vend_id;
```

vend_name	num_prods
Bear Emporium	0
Bears R Us	3
Doll House Inc.	4
Fun and Games	2
Furball Inc.	0
Jouets et ours	0

Note: The fully qualified `Vendors.vend_id` in `GROUP BY` is crucial here. What happens if we used `vend.id` or `Products.vend_id`?

Exercise 6: What's Wrong?

Without running it, what is wrong with the following SQL query?

```
SELECT prod_id, quantity
FROM OrderItems
WHERE quantity = 100
ORDER BY prod_id;
UNION
SELECT prod_id, quantity
FROM OrderItems
WHERE prod_id IN ('BNBG01', 'BNBG02')
ORDER BY prod_id;
```

Exercise 6: What's Wrong? (Solution)

There are two errors:

1. The semicolon (`;`) at the end of the first query signals to SQL to end the query, so `UNION` will not work as written.
2. After removing the semicolon, we cannot have separate `ORDER BY` clauses for each component statement. The `ORDER BY` clause is only allowed at the end of the `UNION`.

Exercise 6: What's Wrong? (Solution)

```sql
SELECT prod_id, quantity
FROM OrderItems
WHERE quantity = 100
UNION
SELECT prod_id, quantity
FROM OrderItems
WHERE prod_id IN ('BNBG01', 'BNBG02')
ORDER BY prod_id;
```

prod_id	quantity
BNBG01	10
BNBG01	100
BNBG01	250
BNBG02	10
BNBG02	100

prod_id	quantity
BNBG02	250
BNBG03	100
BR01	100
BR03	100

Follow-up Question: How can we equivalently write this query as a single SELECT statement?

Exercise 6: What's Wrong? (Follow-Up Solution)

```sql
SELECT prod_id, quantity
FROM OrderItems
WHERE quantity = 100 OR prod_id IN ('BNBG01', 'BNBG02')
ORDER BY prod_id;
```

prod_id	quantity
BNBG01	100
BNBG01	10
BNBG01	250
BNBG02	100
BNBG02	10
BNBG02	250
BNBG03	100
BR01	100
BR03	100

Exercise 7: The Best Customers

In a previous exercise (Chapter 5 Exercise 7), we wanted to identify the best customers by how much they have spent.

Using a simple query, we found only the order numbers for all orders with a total price of at least $1000, sorted by order number.

```
SELECT order_num,
       SUM(item_price * quantity) AS total_price
FROM OrderItems
GROUP BY order_num
HAVING SUM(item_price * quantity) >= 1000
ORDER BY order_num;
```

order_num	total_price
20005	1648.0
20007	1696.0
20009	1867.5

Write a SQL statement that returns the names of the customers who placed orders totaling at least $1000, sorted by customer name.

Exercise 7: The Best Customers (Solution 1)

```sql
-- Equijoin syntax
SELECT cust_name, SUM(item_price * quantity) AS total_price
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
  AND Orders.order_num = OrderItems.order_num
GROUP BY cust_name
HAVING SUM(item_price * quantity) >= 1000
ORDER BY cust_name;
```

cust_name	total_price
Fun4All	2025.6
Village Toys	3515.5

Follow-up Question: How can we equivalently write this query as an explicit join?

Exercise 7: The Best Customers (Solution 2)

As an explicit inner join:

```sql
-- ANSI INNER JOIN syntax
SELECT cust_name, SUM(item_price * quantity) AS total_price
FROM Customers
INNER JOIN Orders
ON Customers.cust_id = Orders.cust_id
INNER JOIN OrderItems
ON Orders.order_num = OrderItems.order_num
GROUP BY cust_name
HAVING SUM(item_price * quantity) >= 1000
ORDER BY cust_name;
```

cust_name	total_price
Fun4All	2025.6
Village Toys	3515.5

Conditional Statements

Conditional Statements

A common scenario when exploring or modeling data is to categorize, filter, or transform the data into more informative or usable ways.

Thus far, we have used logical operators in `WHERE` and `HAVING` clauses to filter rows, similar to logical indexing in R. These allow us to subset data based on logical conditions.

But what if we want to modify or compute values conditionally rather than just filter rows?

For example, suppose we want to assign labels like "High", "Medium", or "Low" to customers based on how much they have spent. We want to show different output depending on (i.e., conditioned on) a value in a column.

Is there an equivalent of an if-else statement in SQL, where we can perform different operations based on a condition?

The CASE Statement

The equivalent of an if-else statement in SQL is the **CASE** statement. The basic syntax rules for the CASE statement are shown as follows:

```
CASE
   WHEN condition_1 THEN result_1
   WHEN condition_2 THEN result_2
   ELSE result_else
END;
```

- The CASE keyword must be followed by at least one set of **WHEN** and **THEN** clauses that specify, respectively, a condition and a result to return when the condition is true.
- An **ELSE** clause is optional but can be used to capture values that are not specified in the earlier WHEN/THEN clauses. If no ELSE is provided, the statement returns NULL.
- The CASE statement must end in an END keyword to close the statement.

Example: Binary Variable

Suppose we want to create a binary indicator for which products cost more than $8 and which cost less than $8. We can use:

```
SELECT prod_name,
       CASE
         WHEN prod_price > 8 THEN 1
         ELSE 0
       END AS over_8
FROM Products
LIMIT 6;
```

prod_name	over_8
8 inch teddy bear	0
12 inch teddy bear	1
18 inch teddy bear	1
Fish bean bag toy	0
Bird bean bag toy	0
Rabbit bean bag toy	0

Example: CASE with Aggregate Functions

The `CASE` statement can also be used directly inside aggregate functions or in other clauses.

For example, to find the number of products that cost over $8, we can use:

```
SELECT SUM(CASE
             WHEN prod_price > 8 THEN 1
             ELSE 0
           END) AS over_8
FROM Products;
```

over_8
4

Note: Writing this using a simple `WHERE prod_price > 8` clause is likely shorter here, but more complex `WHERE` clauses with many `OR` conditions might be clearer and more efficient as a `CASE`.

Example: The CASE Statement

Suppose we want to classify products into categories based on their price. We can use:

```
SELECT prod_name,
       CASE
         WHEN prod_price < 5 THEN 'Bargain'
         WHEN prod_price BETWEEN 5 AND 10 THEN 'Affordable'
         ELSE 'Expensive'
       END AS price_category
FROM Products
LIMIT 6;
```

prod_name	price_category
8 inch teddy bear	Affordable
12 inch teddy bear	Affordable
18 inch teddy bear	Expensive
Fish bean bag toy	Bargain
Bird bean bag toy	Bargain
Rabbit bean bag toy	Bargain

Short-Circuit Evaluation

The previous example can equivalently be written as:

```
SELECT prod_name,
       CASE
         WHEN prod_price < 5  THEN 'Bargain'
         WHEN prod_price < 10 THEN 'Affordable'
         ELSE 'Expensive'
       END AS price_category
FROM Products
LIMIT 6;
```

Notice that the products that cost less than 5 are also less than 10, so such items would satisfy more than one condition. Why does this still work?

The `CASE` statement uses **short-circuit evaluation**: The conditions are evaluated in order, and as soon as one condition is true, it stops evaluating and immediately returns the corresponding result.

Simple CASE Statement

In the special case of `CASE` in which one expression is compared (with equality) to a list of expressions, we can use a **simple `CASE` statement**.

The simple `CASE` syntax is shown below:

```
CASE case_expression
   WHEN when_expression_1 THEN result_1
   WHEN when_expression_2 THEN result_2
   ELSE result_else
END;
```

This is equivalent to the general version (also called **searched `CASE`**):

```
CASE
   WHEN case_expression = when_expression_1 THEN result_1
   WHEN case_expression = when_expression_2 THEN result_2
   ELSE result_else
END;
```

Both versions use short-circuit evaluation.

Example: Simple CASE Versus Searched CASE

For example, the query below using a simple CASE statement

```
SELECT prod_name,
       CASE prod_price
         WHEN 3.49 THEN 'Bargain'
         WHEN 4.99 THEN 'Affordable'
         ELSE 'Expensive'
       END AS price_category
FROM Products;
```

is equivalent to the searched CASE statement

```
SELECT prod_name,
       CASE
         WHEN prod_price = 3.49 THEN 'Bargain'
         WHEN prod_price = 4.99 THEN 'Affordable'
         ELSE 'Expensive'
       END AS price_category
FROM Products;
```

The IF and IIF Functions

For single conditions, some SQL dialects have shorthand functions that make writing conditional statements shorter.

In SQLite (after version 3.32.0), the `IIF()` function has the following syntax (identical to the `ifelse()` function in R):

```
IIF(expression, true_expression, false_expression);
```

If `expression` is true, the `IIF()` function returns `true_expression`, otherwise it returns `false_expression`.

The `IIF()` function is equivalent to the `CASE` statement

```
CASE
   WHEN expression THEN true_expression
   ELSE false_expression
END;
```

In MySQL, the equivalent function is `IF()`, but this type of function is not standard or consistent across SQL dialects.

CASE Closed

Some general tips for using conditional statements:

- The searched `CASE` syntax is the most common form to learn and use. The simple `CASE` syntax is helpful to know but less commonly used in practice.
- `CASE` statements (and `IIF()`/`IF()` functions) can be nested, though nesting too much will be complex and hard to read.
- Be careful with the the order of the conditions, as short-circuit evaluation can cause unintended results if the conditions are ordered incorrectly.
- Though `IIF()` and `IF()` are shorter to write than the equivalent `CASE` statement, use them with caution, as they are not standard across SQL dialects. The `CASE` statement is ANSI SQL that works in all dialects.