Practice with Basic SQL Queries

Chapter 5

Michael Tsiang

Stats 167: Introduction to Databases

UCLA

Basic SQL Exercises

Wildcard Filtering

Basic SQL Exercises

Review: Clauses of a SELECT Statement

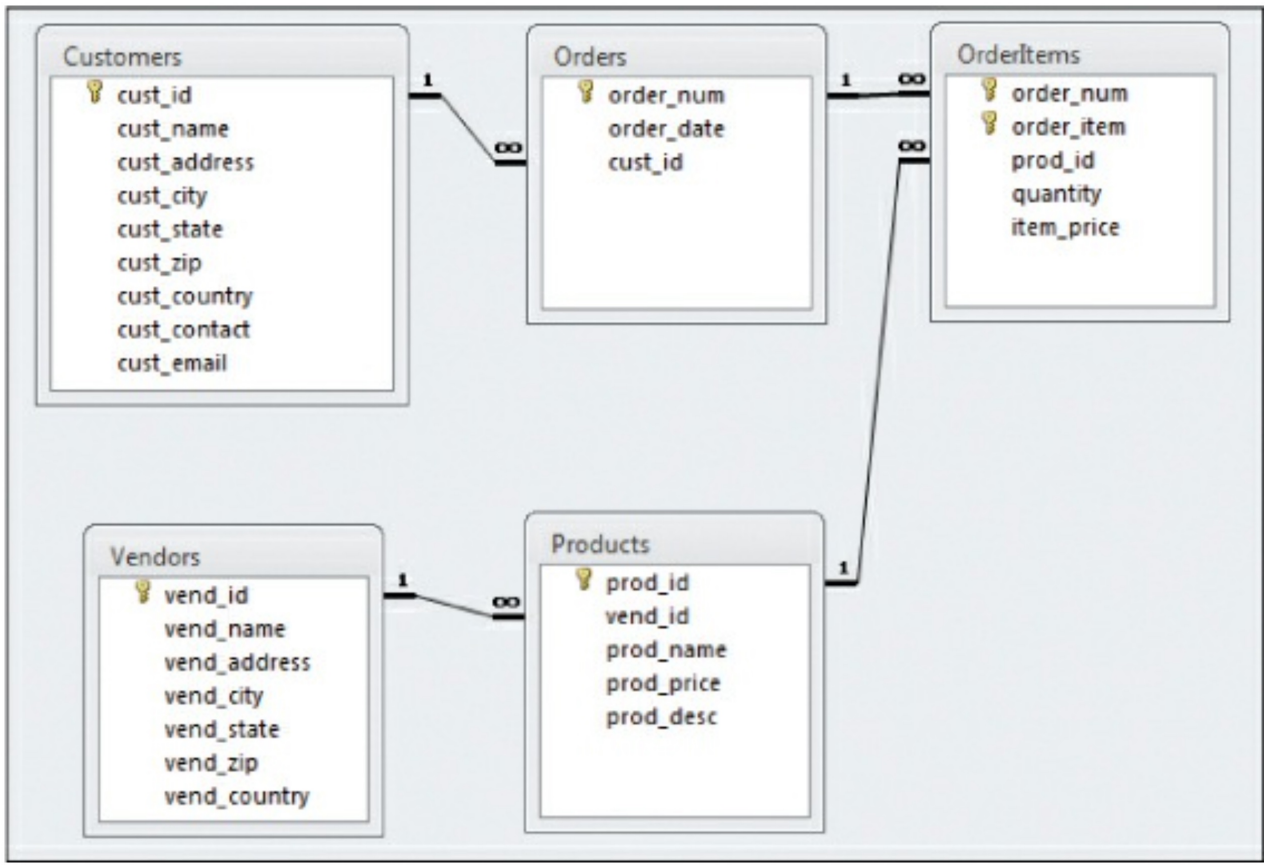
Recall: The `SELECT` statement in SQL is used to retrieve columns of data from one or more tables. The main clauses of a `SELECT` statement are summarized below.

Clause	Description	Required
`SELECT`	Columns or expressions to be returned	Yes
`FROM`	Table to retrieve data from	Only if selecting data from a table
`WHERE`	Row-level filtering	No
`GROUP BY`	Group specification	Only if calculating aggregates by group
`HAVING`	Group-level filtering	No
`ORDER BY`	Output sort order	No
`LIMIT`	Limits number of rows	No

Now that we have an overview of the components of `SELECT`, 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: Product Names and Prices

Write a SQL statement that returns the product name and price for all products priced between $3 and $6 (inclusive).

Use an `AND`, and sort the results by price.

Exercise 1: Product Names and Prices (Solution 1)

```sql
SELECT prod_name, prod_price
FROM products
WHERE prod_price >= 3 AND prod_price <= 6
ORDER BY prod_price;
```

prod_name	prod_price
Fish bean bag toy	3.49
Bird bean bag toy	3.49
Rabbit bean bag toy	3.49
Raggedy Ann	4.99
8 inch teddy bear	5.99

Follow-up Question: How can we write this without `AND`?

Exercise 1: Product Names and Prices (Solution 2)

```sql
SELECT prod_name, prod_price
FROM products
WHERE prod_price BETWEEN 3 AND 6
ORDER BY prod_price;
```

prod_name	prod_price
Fish bean bag toy	3.49
Bird bean bag toy	3.49
Rabbit bean bag toy	3.49
Raggedy Ann	4.99
8 inch teddy bear	5.99

Exercise 2: What's Wrong?

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

```
SELECT vend_name
FROM Vendors
ORDER BY vend_name
WHERE vend_country = 'USA' AND vend_state = 'CA'
```

Exercise 2: What's Wrong? (Solution)

The `ORDER BY` needs to be placed last in the statement.

The corrected statement:

```sql
SELECT vend_name
FROM Vendors
WHERE vend_country = 'USA' AND vend_state = 'CA'
ORDER BY vend_name
```

vend_name
Doll House Inc.

Exercise 3: Sale Pricing

Suppose our example store is running a sale and all products are 10% off.

Write a SQL statement that returns the product ID, the (original) product price, and the sale price for all products sold.

Exercise 3: Sale Pricing (Solution)

```sql
SELECT prod_id, prod_price,
       prod_price * 0.9 AS sale_price
FROM Products;
```

prod_id	prod_price	sale_price
BR01	5.99	5.391
BR02	8.99	8.091
BR03	11.99	10.791
BNBG01	3.49	3.141
BNBG02	3.49	3.141
BNBG03	3.49	3.141
RGAN01	4.99	4.491
RYL01	9.49	8.541
RYL02	9.49	8.541

Follow-up Question: What if we wanted to round the sale price to two decimal places?

Exercise 3: Sale Pricing (Rounded Solution)

The `ROUND()` function can be used to round the results.

```
SELECT prod_id, prod_price,
       ROUND(prod_price * 0.9, 2) AS sale_price
FROM Products;
```

prod_id	prod_price	sale_price
BR01	5.99	5.39
BR02	8.99	8.09
BR03	11.99	10.79
BNBG01	3.49	3.14
BNBG02	3.49	3.14
BNBG03	3.49	3.14
RGAN01	4.99	4.49
RYL01	9.49	8.54
RYL02	9.49	8.54

Exercise 4: Total Items Ordered

Write a SQL statement to determine the total number of items sold (across all orders in the data).

Exercise 4: Total Items Ordered (Solution)

```sql
SELECT SUM(quantity) AS items_ordered
FROM OrderItems;
```

items_ordered
1430

Follow-up Question: How would we modify this query to determine the total number of product item `BR01` sold?

Exercise 4: Total Items Ordered (Follow-Up Solution)

```sql
SELECT SUM(quantity) AS items_ordered
FROM OrderItems
WHERE prod_id = 'BR01';
```

items_ordered
120

Exercise 5: Most Expensive Item

Write a SQL statement to determine the price of the most expensive product that costs no more than $10.

Exercise 5: Most Expensive Item (Solution)

```sql
SELECT MAX(prod_price) AS max_price
FROM Products
WHERE prod_price <= 10;
```

max_price
9.49

Exercise 6: The Largest Orders

We appreciate customer loyalty, so it's important to identify the best customers.

Write a SQL statement to return the order number for all orders of at least 100 items.

Exercise 6: The Largest Orders (Solution)

```sql
SELECT order_num
FROM OrderItems
GROUP BY order_num
HAVING SUM(quantity) >= 100
ORDER BY order_num;
```

order_num
20005
20007
20009

Follow-up Question: To make the output more informative, how can we include the total number of items ordered for each order number?

Exercise 6: The Largest Orders (Follow-Up Solution)

```sql
SELECT order_num, SUM(quantity) AS total_quantity
FROM OrderItems
GROUP BY order_num
HAVING SUM(quantity) >= 100
ORDER BY order_num;
```

order_num	total_quantity
20005	200
20007	400
20009	750

Note: We can use the alias `total_quantity` in the `HAVING` clause (i.e., `HAVING total_quantity >= 100`) and it would still work in SQLite. However, not every DBMS will allow aliases in `HAVING`, as the `GROUP BY` and `HAVING` clauses are evaluated *before* `SELECT`. It is best practice not to use aliases in `HAVING`.

Exercise 7: The Most Expensive Orders

Another way to determine the best customers is by how much they have spent.

Write a SQL statement to return the order number for all orders with a total price of at least $1000. Sort the results by order number.

Exercise 7: The Most Expensive Orders (Solution)

```sql
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

Exercise 8: What's Wrong (Part 2)?

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

```sql
SELECT order_num, COUNT(*) AS items
FROM OrderItems
GROUP BY items
HAVING COUNT(*) >= 3
ORDER BY items, order_num;
```

Exercise 8: What's Wrong (Part 2)? (Solution)

The clause `GROUP BY items` is incorrect. As noted earlier (Exercise 6), `GROUP BY` is evaluated before `SELECT`. Thus `GROUP BY` must be an actual column, not the one being used to perform the aggregate calculations.

`GROUP BY order_num` would be allowed.

```
SELECT order_num, COUNT(*) AS items
FROM OrderItems
GROUP BY order_num
HAVING COUNT(*) >= 3
ORDER BY items, order_num;
```

order_num	items
20006	3
20009	3
20007	5
20008	5

Wildcard Filtering

Filtering Operators

Recall: The `WHERE` and `HAVING` clauses support many conditional operators for data filtering:

Operator	Description
=	Equality
<>	Nonequality
!=	Nonequality
<	Less than
<=	Less than or equal to
!<	Not less than
>	Greater than
>=	Greater than or equal to
!>	Not greater than
`BETWEEN`	Between two specified values
`IS NULL`	Is a `NULL` (i.e., missing) value

Filtering on Unknown Values

All the previous operators for data filtering we have seen so far filter data against *known* values. However, filtering data using known values does not always work.

For example, how can we search for all products that contain the text "bean bag" in the product name?

A literal match of the text "bean bag" would not return any results, since "bean bag" is only a partial name match.

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name = 'bean bag';
```

prod_id	prod_name

Is there another operator that can match on "bean bag" anywhere in the product name?

The LIKE Operator

In order to perform more nuanced filtering, we need to use **wildcard filtering** to create a search pattern that can be compared against our data.

- A **wildcard** is a special character used to match parts of a value.
- A **search pattern** is a search condition made up of literal text, wildcard characters, or any combination of them.

To use wildcards in search clauses, we need the `LIKE` operator. `LIKE` instructs the DBMS to compare a search pattern using a wildcard match rather than a literal equality match.

The Percent Sign (%) Wildcard

The most frequently used wildcard is the **percent sign (%)**.

Within a search string, % means *match any number of occurrences of any character* (i.e., 0 or more occurrences).

For example, to find all products that start with the word "Fish", we can use:

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE 'Fish%';
```

prod_id	prod_name
BNBG01	Fish bean bag toy

Note: Depending on the DBMS and how it is configured, searches may be case sensitive. SQLite by default is *not* case sensitive, so `'fish%'` will work here too.

Exercise: Finding Bean Bag Toys

Now that we have learned the `LIKE` operator, how can we search for all products that contain the text "bean bag" anywhere in the product name?

Hint: Why is the search pattern `'bean bag%'` insufficient?

Exercise: Finding Bean Bag Toys (Solution)

```sql
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '%bean bag%';
```

prod_id	prod_name
BNBG01	Fish bean bag toy
BNBG02	Bird bean bag toy
BNBG03	Rabbit bean bag toy

The Underscore (_) Wildcard

Another useful wildcard is the **underscore (_)**.

The underscore is like % in that it matches any character, except that it matches only *single* characters.

For example, to find all teddy bear products that are double digit inches tall, we can use:

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '__ inch teddy bear';
```

prod_id	prod_name
BR02	12 inch teddy bear
BR03	18 inch teddy bear

Comparing % and _

Note the difference:

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '_ inch teddy bear';
```

prod_id	prod_name
BR01	8 inch teddy bear

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '% inch teddy bear';
```

prod_id	prod_name
BR01	8 inch teddy bear
BR02	12 inch teddy bear
BR03	18 inch teddy bear

The ESCAPE Clause

To search for a literal % or _ in the pattern, we would need to **escape** the special property of the wildcard characters. To do this, we would need to escape the character with \ inside the pattern and then add the \ in an **ESCAPE** clause.

For example, the following query would match any product name that starts with 100%:

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '100\%%' ESCAPE '\';
```

LIKE is not REGEX

The `LIKE` operator and wildcards allow for flexible queries and opens many possibilities for filtering results.

However, the `LIKE` operator has its limitations:

- The `LIKE` operator does not support general regular expressions (regex). SQLite has a `REGEXP` operator, but it requires a user-defined function or extension to use. Other DBMSs have more powerful regex functions.
- The character set or bracket `[]` wildcard can be used with `LIKE` for some DBMSs but not all. Microsoft SQL Server supports character sets but MySQL, Oracle, DB2, and SQLite do not.

The GLOB Operator

The `GLOB` operator (stands for global) is another pattern matching operator with some distinctions from `LIKE`.

- `GLOB` uses `*` instead of `%`.
- `GLOB` uses `?` instead of `_`.
- `GLOB` is case sensitive, while `LIKE` is not case sensitive.
- `GLOB` can use character sets `[a-z]` and negated sets `[^a-z]`.
- `GLOB` cannot escape characters, while `LIKE` can use `ESCAPE`.

Example: GLOB

For example, to find all products that do not start with a number, we can use:

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name GLOB '[^1-9]*';
```

prod_id	prod_name
BNBG01	Fish bean bag toy
BNBG02	Bird bean bag toy
BNBG03	Rabbit bean bag toy
RGAN01	Raggedy Ann
RYL01	King doll
RYL02	Queen doll

With Great Power (Everything is Optimization)

Wildcard filtering can be very powerful and useful, but wildcard searches can take much longer to process than more basic types of searching/filtering.

Some general tips for using wildcards:

- Do not overuse wildcards. If another search operator will work, use it instead.
- Try not to use wildcards at the beginning of a search pattern unless necessary. Search patterns that begin with wildcards are the slowest to process.
- Pay careful attention to the placement of wildcard symbols. Misplaced wildcards can produce unintended results.