SQL Interview Questions

Our schema

We'll be going through six questions covering topics like query performance, joins, and SQL injection. They'll refer to the same database for cakes, customers, and orders at a bakery. Here's the schema:

```sql
CREATE TABLE cakes (
    cake_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    flavor VARCHAR(100) NOT NULL
);

CREATE TABLE customers (
    customer_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    phone VARCHAR(15),
    street_address VARCHAR(255),
    city VARCHAR(255),
    zip_code VARCHAR(5),
    referrer_id INT,
    FOREIGN KEY (referrer_id) REFERENCES customers (customer_id)
);

CREATE TABLE orders (
    order_id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    cake_id INT NOT NULL,
    customer_id INT,
    pickup_date DATE NOT NULL,
    FOREIGN KEY (cake_id) REFERENCES cakes (cake_id),
    FOREIGN KEY (customer_id) REFERENCES customers (customer_id)
);
```

We'll be using MySQL for consistency.

Want to get set up with sample data? Here's how to get four cakes, a million customers, and a million orders:

In your terminal, download our script and start up MySQL:

```
$ curl -O https://s3-us-west-2.amazonaws.com/icpublic/bakery_schema_and_data.sql && mysql.server sta
```

Then run our script to set up the `BAKERY` database and insert data:

```
> source bakery_schema_and_data.sql;
```

> If you want to come back to the data again and you already downloaded and ran the script:
>
> 1. In terminal, start up MySQL: `mysql.server start && mysql -u root`
> 2. In the MySQL shell, use the bakery database: `USE BAKERY;`

Alright, let's get started.

How can we make this query faster?

We want the order ID of every order in January and February, 2017. This is the query we've got so far:

SQL
```
SELECT order_id FROM orders WHERE DATEDIFF(orders.pickup_date, '2017-03-01') < 0;
-- 161314 rows in set (0.25 sec)
```

Answer

First, we could consider adding an index on `pickup_date`. This'll make our *inserts* less efficient, but will drastically make this query faster.

> Bringing up the **tradeoffs** of your suggestions will impress your interviewer. Excellent answers always consider the bigger picture—side effects or downsides, and any alternative solutions.

Let's try it:

SQL
```
ALTER TABLE orders ADD INDEX (pickup_date);

SELECT order_id FROM orders WHERE DATEDIFF(orders.pickup_date, '2017-03-01') < 0;
-- 161314 rows in set (0.24 sec)
```

Whoa! What happened? We added an index but didn't get an improvement.

This is because we're using the `DATEDIFF` function in the `WHERE` clause. ***Functions* evaluate for every row in a table without using the index!**

Easy fix here—we can just compare the pickup date and March 1 directly:

SQL
```
SELECT order_id FROM orders WHERE orders.pickup_date < '2017-03-01';
-- 161314 rows in set (0.07 sec)
```

There we go, about 0.3 seconds down to 0.07 seconds.

How can we get the *n*th highest number of orders a single customer has?

We made a view of the number of orders each customer has:

```sql
CREATE VIEW customer_order_counts AS
SELECT customers.customer_id, first_name, count(orders.customer_id) AS order_count
FROM customers LEFT OUTER JOIN orders
ON (customers.customer_id = orders.customer_id)
GROUP BY customers.customer_id;
```

So for example, Nancy has 3 orders:

```sql
SELECT * FROM customer_order_counts ORDER BY RAND() LIMIT 1;

/*
    +-------------+------------+-------------+
    | customer_id | first_name | order_count |
    +-------------+------------+-------------+
    |        9118 | Nancy      |           3 |
    +-------------+------------+-------------+
*/
```

Answer

Let's start by solving a simpler problem—how would we just get the **highest** number of orders?

Pretty trivial since we have a built-in function:

```sql
SELECT MAX(order_count) FROM customer_order_counts;
-- 1 row in set (1.89 sec)
```

Now can we adapt this to get the **second highest** number of orders?

Well, if we can get the highest order count, we can get "the highest order count that's not the *highest* order count":

```sql
SELECT MAX(order_count) FROM customer_order_counts WHERE order_count NOT IN (
SELECT MAX(order_count) FROM customer_order_counts);
-- 1 row in set (3.89 sec)
```

This works, but it's pretty slow. Any ideas for a faster query?

We could also *sort* the `order_counts`. Then we don't have to scan the whole table *twice*. We just jump down to the second row:

```sql
SELECT order_count FROM customer_order_counts ORDER BY order_count DESC LIMIT 1, 1;
-- 1 row in set (1.93 sec)
```

> If `LIMIT` has one argument, that argument is the number of rows to return starting with the first row. With 2 arguments, the *first* argument is the row offset and the *second* argument is the number of rows to return. So in our query with "1, 1" we're saying "starting one row down from the top, give us one row."

Now, how do we get the ***nth* highest** order count?

Easy—we just change the row offset in our `LIMIT` clause:

```sql
SELECT order_count FROM customer_order_counts ORDER BY order_count DESC LIMIT N-1, 1;
```

As a bonus, how would you do this with pure SQL, not relying on MySQLs handy `LIMIT` clause? It's tricky!

What ways can we use wildcard characters in `LIKE` clauses?

Answer

`LIKE` lets you use two wildcard characters, "%" and "_".

"%" matches any amount of characters (including zero characters). So if we want all our customers whose last name starts with "A", we could query:

```sql
SELECT customer_id, last_name FROM customers WHERE last_name LIKE 'a%';
```

(We could use the `BINARY` keyword if we wanted a case-sensitive comparison. It would treat the string we're comparing as a binary string, so we'd compare bytes instead of characters.)

"_" matches exactly one character. If we want all our customers who live on the 200 block of Flatley Avenue in Dover, we could query:

```sql
SELECT first_name, street_address FROM customers
WHERE street_address LIKE '2__ Flatley Avenue' AND city = 'Dover';
```

And some databases (like SQL Server, but not MySQL or PostgreSQL) support sets or ranges of characters. So we could get every customer whose city starts with either "m" or "d":

SQL
```sql
SELECT customer_id FROM customers WHERE city LIKE '[md]%';
```

Or whose last name starts with any character in the *range* “a” through “m” (“a”, “b”, “c”...“k”, “l”, “m”):

SQL
```sql
SELECT customer_id FROM customer WHERE last_name LIKE '[a-m]%'
```

Now how can we make *this* query faster?

We're mailing a promotion to all our customers named Sam who live in Dover. Since some customers go by names that aren't exactly Sam, like Samuel or Sammy, we use names *like* Sam. Here's how we find them:

SQL
```sql
SELECT first_name, last_name, street_address, city, zip_code FROM customers
WHERE first_name LIKE '%sam%' AND city = 'Dover';
-- 1072 rows in set (0.42 sec)
```

That's pretty slow. How can we speed it up?

Answer

First, do we need to get the city and zip code for every customer? The search is *constructed* using the city Dover, so we *know* the city. And we know that the zip code for Dover is 33220. If we can complete the addresses efficiently somewhere else in our code, there's no reason to get that information from the database for every result.

```sql
SELECT first_name, last_name, street_address FROM customers
WHERE first_name LIKE '%sam%' AND city = 'Dover';
-- 1072 rows in set (0.40 sec)
```

A little better, but only a little.

Let's look at that wildcard % *before* "sam." Wildcards at the beginning of comparisons can slow down performance because instead of just looking at names that *start* with "sam" the query has to look at *every* character in *every* first name.

Do we really need the wildcard % before sam? Should our customers with "sam" *in* their name but *not at the start* of their name, like Rosamond, be included in a Sam Promotion?

Probably not. Let's just try removing the % at the beginning and adding an index on `first_name`:

```sql
ALTER TABLE customers ADD INDEX (first_name);


SELECT first_name, last_name, street_address FROM customers
WHERE first_name LIKE 'sam%' AND city = 'Dover';
-- 1065 rows in set (0.02 sec)
```

0.42 seconds down to 0.02 seconds!

This is a huge improvement. ***But*—these changes are a big deal because we're changing *functionality*.** This isn't just *faster*, it's *different*. Some customers *won't* be getting a promotion in the mail now. The decision of who's included in the promotion would probably be made independent of database performance. But it's always a good idea to look out for wildcard characters at the beginning of a pattern.

What are all the SQL joins?

Answer

First, let's talk about the keywords "inner" and "outer." They're optional—`INNER JOIN` is the same as `JOIN`, and `LEFT OUTER JOIN` is the same as `LEFT JOIN`. These keywords are added for clarity because they make the joins easier to understand conceptually. Some developers leave them out, arguing there's no reason to have extra nonfunctional words in a database query. The most important thing is to be consistent. We'll use them.

Inner joins give *only* the rows where *all* the joined tables have related data. If we inner join our customers and orders, we'll get all the related customers and orders. We *won't* get any customers without orders or any orders without customers.

```sql
SELECT first_name, phone, orders.cake_id, pickup_date
FROM customers INNER JOIN orders
ON customers.customer_id = orders.customer_id;


/*
    +------------+------------+---------+-------------+
    | first_name | phone      | cake_id | pickup_date |
    +------------+------------+---------+-------------+
    | Linda      | 8095550114 |       4 | 2017-10-12  |
    | May        | 8015550130 |       4 | 2017-02-03  |
    | Frances    | 8345550120 |       1 | 2017-09-16  |
    | Mathew     | 8095550122 |       3 | 2017-07-20  |
    | Barbara    | 8015550157 |       2 | 2017-07-07  |
    ...
*/
```

If we wanted the cake *flavor*, not just the cake ID, we could also join the cake table:

```sql
SELECT first_name, phone, cakes.flavor, pickup_date
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id
INNER JOIN cakes ON orders.cake_id = cakes.cake_id;

/*
    +------------+------------+-----------+-------------+
    | first_name | phone      | flavor    | pickup_date |
    +------------+------------+-----------+-------------+
    | Frances    | 8345550120 | Chocolate | 2017-09-16  |
    | Theodore   | 8015550175 | Chocolate | 2017-08-13  |
    | James      | 8015550165 | Chocolate | 2017-10-12  |
    | Kathleen   | 8095550157 | Chocolate | 2017-09-24  |
    | Jennifer   | 8015550153 | Chocolate | 2017-06-22  |
    ...
*/
```

Left outer joins give *all* the rows from the first table, but *only* related rows in the next table. So if we run a left outer join on customers and orders, we'll get *all* the customers, and their orders *if* they have any.

```sql
SELECT cake_id, pickup_date, customers.customer_id, first_name
FROM orders LEFT OUTER JOIN customers
ON orders.customer_id = customers.customer_id
ORDER BY pickup_date;

/*
    +---------+-------------+-------------+------------+
    | cake_id | pickup_date | customer_id | first_name |
    +---------+-------------+-------------+------------+
    |       2 | 2017-01-01  |        NULL | NULL       |
    |       3 | 2017-01-01  |      108548 | Eve        |
    |       1 | 2017-01-01  |      857831 | Neil       |
    |       4 | 2017-01-01  |        NULL | NULL       |
    |       3 | 2017-01-01  |      168516 | Maria      |
    ...
*/
```

Right outer joins include any related rows in the first table, and *all* the rows in the next table. Right outer joining our customers and orders would give the customer if there is one, and then *every* order.

> In our schema, `customer_id` isn't `NOT NULL` on orders. This may be seem unintuitive, but maybe we don't require customers to register with us to place an order, or orders can be associated with other models like `restaurant` or `vendor`. In any case, with our schema, we can have orders without customers.

```sql
SELECT customers.customer_id, first_name, pickup_date
FROM customers RIGHT OUTER JOIN orders
ON customers.customer_id = orders.customer_id
ORDER BY pickup_date;

/*
    +-------------+------------+-------------+
    | customer_id | first_name | pickup_date |
    +-------------+------------+-------------+
    |        NULL | NULL       | 2017-01-01  |
    |      108548 | Eve        | 2017-01-01  |
    |      857831 | Neil       | 2017-01-01  |
    |        NULL | NULL       | 2017-01-01  |
    |        NULL | NULL       | 2017-01-01  |
    ...
*/
```

Right outer joins give the same result as left outer joins with the order of the tables switched:

```sql
SELECT customers.customer_id, first_name, pickup_date
FROM orders LEFT OUTER JOIN customers
ON customers.customer_id = orders.customer_id
ORDER BY pickup_date;

/*
    same results as right outer join we just did!

    +-------------+------------+-------------+
    | customer_id | first_name | pickup_date |
    +-------------+------------+-------------+
    |        NULL | NULL       | 2017-01-01  |
    |      108548 | Eve        | 2017-01-01  |
    |      857831 | Neil       | 2017-01-01  |
    |        NULL | NULL       | 2017-01-01  |
    |        NULL | NULL       | 2017-01-01  |
    ...
*/
```

Full outer joins take *all* the records from *every* table. Related data are combined like the other joins, but no rows from any table are left out. For customers and orders, we'll get all the related customers and orders, *and* all the customers without orders, *and* all the orders without customers.

The standard SQL syntax is:

```sql
SELECT order_id, pickup_date, customers.customer_id, first_name

FROM orders FULL OUTER JOIN customers

ON orders.customer_id = customers.customer_id
```

But MySQL doesn't support full outer joins! No problem, we can get the same result with a `UNION` of left and right outer joins:

```sql
SELECT order_id, pickup_date, customers.customer_id, first_name
FROM orders LEFT OUTER JOIN customers
ON orders.customer_id = customers.customer_id

UNION

SELECT order_id, pickup_date, customers.customer_id, first_name
FROM orders RIGHT OUTER JOIN customers
ON orders.customer_id = customers.customer_id;


/*
    +----------+------------+------------+-----------+
    | order_id | pickup_date | customer_id | first_name |
    +----------+------------+------------+-----------+
    |   900075 | 2017-05-17  |        NULL | NULL       |
    |   900079 | 2017-12-26  |      487996 | Frances    |
    |   900057 | 2017-10-25  |      498546 | Loretta    |
    |     NULL | NULL        |      640804 | Whitney    |
    |     NULL | NULL        |       58405 | Zoe        |
    ...
*/
```

> Using `UNION` or `UNION ALL` with this strategy generally emulates a full outer join. But things get complicated for some schemas, like if a column in the `ON` clause isn't `NOT NULL`.

Cross joins give *every* row from the first table paired with *every row* in the next table, ignoring any relationship. With customers and orders, we'd get *every* customer paired with *every* order. Cross joins are sometimes called **Cartesian joins** because they return the **cartesian product** of data sets—every combination of elements in every set.

This isn't used often because the results aren't usually useful. But sometimes you might actually need every combination of the rows in your tables, or you might need a large table for performance testing. If you cross join 2 tables with 10,000 rows each, you get a table with 100,000,000 rows!

Self joins refer to any join that joins data in the *same* table. For example, some of our customers were referred to our bakery by other customers. We could do a left outer join to get every customer and their referrer if they have one:

```sql
SELECT customer.first_name, referrer.first_name
FROM customers AS customer LEFT OUTER JOIN customers AS referrer
ON customer.referrer_id = referrer.customer_id;

/*
    +------------+------------+
    | first_name | first_name |
    +------------+------------+
    | Tim        | NULL       |
    | Mattie     | Wendy      |
    | Kurtis     | NULL       |
    | Jared      | NULL       |
    | Lucille    | Tim        |
    ...
*/
```

What’s an example of SQL injection and how can we prevent it?

Answer

SQL injection is when a hacker gains access to our database because we used their malicious user input to build a dynamic SQL query.

Let's say we have an input field that takes a phone number, and we use it to build this SQL query:

```java
String sqlText = "SELECT * FROM customers WHERE phone = '" + phoneInput + "';";
```

We’re expecting something like "8015550198" which would neatly build:

```sql
SELECT * FROM customers WHERE phone = '8015550198';
```

But what if a user enters "1' OR 1=1;--"?

Then we’d have:

```sql
SELECT * FROM customers WHERE phone = '1' OR 1=1;--';
```

Which will return the data for every customer because the `WHERE` clause will *always* evaluate to true! (1 always equals 1, and the "--" comments out the last single quotation mark.)

With the right input and queries, SQL injection can let hackers create, read, update and destroy data.

So to prevent SQL injection, we'll need to look at how we build and run our SQL queries. And we can think about some smart technical design in our application.

Here are five ways to protect ourselves:

1. Use stored procedures or prepared SQL statements. So do *not* build dynamic SQL. This is the most effective way to prevent SQL injection.

For example, we could build a prepared statement:

```java
import java.sql.*;

// register MySQL JDBCdriver
Class.forName("com.mysql.cj.jdbc.Driver").newInstance();

// connect to the database server
try (Connection connection = DriverManager.getConnection("jdbc:mysql://localhost/bakery?user=root"))

    // create prepared statement
    String statement = "SELECT * FROM customers WHERE phone = ?";
    PreparedStatement preparedStatement = connection.prepareStatement(statement);

    // the first argument in setter functions is an integer n
    // corresponding to the nth ? in the statement
    preparedStatement.setString(1, inputPhone);

    // execute statement
    try (ResultSet results = preparedStatement.executeQuery()) {

        // process resulting rows
    }
}
```

Or we could build a stored procedure `get_customer_from_phone()` with a string parameter `inputPhone`:

```sql
DELIMITER //
CREATE PROCEDURE get_customer_from_phone
(IN input_phone VARCHAR(15))
BEGIN
    SELECT * FROM customers
    WHERE phone = input_phone;
END //
DELIMITER ;
```

which we could call like this:

```java
import java.sql.*;

// register MySQL JDBCdriver
Class.forName("com.mysql.cj.jdbc.Driver").newInstance();

// connect to the database server
try (Connection connection = DriverManager.getConnection("jdbc:mysql://localhost/bakery?user=root"))

    // prepare callable statement
    CallableStatement preparedStatement = connection.prepareCall(
            "{call get_customer_from_phone(?)}");

    // the first argument in setter functions is an integer n
    // corresponding to the nth ? in the statement
    callableStatement.setString(1, inputPhone);

    // execute statement
    try (ResultSet results = callableStatement.executeQuery()) {

        // process resulting rows
    }
}
```

2. Validate the type and pattern of input. If you know you're looking for specific data—like an ID, name, or email address—validate any user input based on type, length, or other attributes.

For example, here’s one way we could validate a phone number:

Java ▾

```java
import java.util.regex.Pattern;

public static boolean isValidPhone(String phoneNumber) {

    // check phone number for null or empty
    if (phoneNumber == null || phoneNumber.isEmpty()) {
        return false;
    }

    // contains only valid phone characters
    // has exactly 10 digits
    return Pattern.matches(
        "^\\(?([0-9]{3})\\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$",
        phoneNumber);
}
```

3. Escape special characters like quotes. This method is a quick and easy way to reduce the chances of SQL injection, but it's not fully effective.

For example, let's say we want to escape backslashes, single and double quotes, new lines (`\n` and `\r`), and null (`\0`):

Java ▾

```java
private static final class EscapeRule {
    EscapeRule(Character from, String to) {
        this.from = from;
        this.to = to;
    }

    Character from;
    String to;
}

private static final EscapeRule[] ESCAPE_RULES = new EscapeRule[] {
    new EscapeRule('\\', "\\\\"),
    new EscapeRule('\0', "\\0"),
    new EscapeRule('\n', "\\n"),
    new EscapeRule('\r', "\\r"),
    new EscapeRule('\'',  "\\'"),
    new EscapeRule('\"', "\\\""),
};

private static final HashMap<Character, String> charToReplacement = new HashMap<>();

public static String escapeInput(String input)
{
    // fill map with replacement rules, if didn't so yet
    synchronized(charToReplacement) {
        if(charToReplacement.size() < ESCAPE_RULES.length) {
            for (EscapeRule rule : ESCAPE_RULES) {
                charToReplacement.put(rule.from, rule.to);
            }
        }
    }

    StringBuilder result = new StringBuilder();

    // convert each character
    for (int i = 0, n = input.length(); i < n; ++i) {
        char character = input.charAt(i);
        String replacement = charToReplacement.get(character);
        if (replacement != null) {
            result.append(replacement);
        } else {
            result.append(character);
        }
    }

    return result.toString();
}
```

(See table 10.1 in the MySQL String Literals docs (http://dev.mysql.com/doc/refman/5.7/en/string-literals.html) for a full list of special character escape sequences.)

In Java, you can use the OWASP Enterprise Security API (ESAPI), an open-source third-party library with a MySQLCodec class for SQL escaping.

> Apache Commons *used* to have a `StringEscapeUtils.escapeSql()` function, but they removed it in lang3 because it "was a misleading method, only handling the simplest of possible SQL cases. As SQL is not Lang's focus, it didn't make sense to maintain this method." (Search for `StringEscapeUtils.escapeSql` in What's new in Commons Lang 3.0? (https://commons.apache.org/proper/commons-lang/article3_0.html))

When we escape our input, now our query will be:

```
SELECT * FROM customers WHERE phone = '1\' OR 1=1;--';
```

which isn't a valid query.

4. Limit database privileges. Application accounts that connect to the database should have as few privileges as possible. It's unlikely, for example, that your application will ever have to delete a table. So don't allow it.

5. Don't display database error messages to users. Error messages contain information that could tell hackers a lot of information about your data. Best practise is to give generic database error messages to users, and log detailed errors where developers can access them. Even better, send an *alert* to the dev team when there’s an error.

What's next?

Check out our mock coding interview questions (/next).

They mimic a real interview by offering hints when you're stuck or you're missing an optimization.

Try some questions now ➡