Make your SQL safer

SQL can be a beautiful thing, but it can also be a dangerous thing. If you're using SQL to access a database for an app that's used by hundreds or thousands or even millions of users, you need to be careful - because you could accidentally damage or erase all that data. There are various techniques you can use to make your SQL safer, however.

Avoiding bad updates/deletes

Before you issue an UPDATE, run a SELECT with the same WHERE to make sure you're updating the right column and row.

For example, before running:

```
UPDATE users SET deleted = true WHERE id = 1;
```

You could run:

```
SELECT id, deleted FROM users WHERE id = 1;
```

Once you decide to run the update, you can use the LIMIT operator to make sure you don't accidentally update too many rows:

```
UPDATE users SET deleted = true WHERE id = 1 LIMIT 1;
```

Or if you were deleting:

```
DELETE users WHERE id = 1 LIMIT 1;
```

Using transactions

When we issue a SQL command that changes our database in some way, it starts what is called a "transaction." A transaction is a sequence of operations treated as a single logical piece of work (like a bank transaction), and in the world of databases, a transaction must comply to the ACID principles to make sure the operations are processed reliably.

Whenever we issue a command like CREATE, UPDATE, INSERT, or DELETE, we are automatically starting a transaction. However, if we want, we can also wrap up multiple commands inside a bigger transaction. It may be that we only want an UPDATE to go through if another UPDATE goes through as well, so we want to put both of them in the same transaction.

In that case, we can wrap the commands in BEGIN TRANSACTION and COMMIT:

```
BEGIN TRANSACTION;
UPDATE people SET husband = "Winston" WHERE user_id = 1;
UPDATE people SET wife = "Winnefer" WHERE user_id = 2;
COMMIT;
```

If the database is unable to issue both those UPDATE commands for some reason, then it will rollback the transaction and leave the database how it was when it started.

We also use transactions when we want to make sure that all of our commands operate on the same view of the data - when we want to ensure that no other transactions operate on that same data while the sequence of commands is running. When you're looking at a sequence of commands you want to run, ask yourself what would happen if another user issued commands at the same time. Could your data end up in a weird state? In that case, you should run in a transaction.

For example, the following commands create a row denoting that a user earned a badge, and then update the user's recent activity to describe that:

```
INSERT INTO user_badges VALUES (1, "SQL Master", "4pm");
UPDATE user SET recent_activity = "Earned SQL Master badge" WHERE id = 1;
```

At the same time, another user or process might be awarding the user a second badge:

```
INSERT INTO user_badges VALUES (1, "Great Listener", "4:05pm");
UPDATE user SET recent_activity = "Earned Great Listener badge" WHERE id = 1;
```

These commands could now actually be issued in this order:

```
INSERT INTO user_badges VALUES (1, "SQL Master");

INSERT INTO user_badges VALUES (1, "Great Listener");

UPDATE user SET recent_activity = "Earned Great Listener badge" WHERE id = 1;

UPDATE user SET recent_activity = "Earned SQL Master badge" WHERE id = 1;
```

Their recent activity would now be "Earned SQL Master badge" even though the most recently inserted badge was "Great listener". That's not the end of the world, but it's also probably not what we expected.

Instead, we could run those in a transaction, to guarantee that no other transactions happen in the middle:

```
BEGIN TRANSACTION;

INSERT INTO user_badges VALUES (1, "SQL Master");

UPDATE user SET recent_activity = "Earned SQL Master badge" WHERE id = 1;

COMMIT;
```

Making backups

You should definitely follow all those tips, but sometimes mistakes still happen. Because of that, most companies make backups of their databases - on an hourly, daily, or weekly basis, depending on the size of the database and space available. When something bad happens, they can then import data from the old database for whichever tables were damaged or lost. The data may end up a little outdated, but outdated data is often better than no data at all.

Replication

A related approach is replication - always storing multiple copies of the databases in different places. If for some reason a particular copy of the database is unavailable (like lightning hit the building that it's in, which

has actually happened to me!), then the query can be sent to another copy of the database which is hopefully still available. If the data is very important, then it should probably be replicated to ensure availability. For example, if a doctor is trying to pull up a list of a patient's allergies to determine how to treat them in an emergency situation, then they can't afford to wait for engineers to get the data out of a backup, they need it immediately.

However, it is a lot more effort to replicate databases and it often means slower performance since write operations have to be performed in all of them, so companies must decide whether the benefits of replication are worth the costs, and investigate the best way of setting it up for their environment.

Granting privileges

Many database systems have users and privileges built into them, because they are stored on a server and accessed by multiple users. There is no concept of user/privilege in the SQL scripts on Khan Academy, because SQLite is typically used in a single-user scenario, and thus you can write to it as long as you have access to the drive it's stored on.

But if you are using a database system on a shared server one day, then you should make sure to set up users and privileges properly from the beginning. As a general rule, there should be only a few users that have full access to the database (like backend engineers), since it can be so dangerous.

For example, here's how we can give full access to a particular user:

```
GRANT FULL ON TABLE users TO super_admin;
```

And here's how we can give only SELECT access to a different user:

```
GRANT SELECT ON TABLE users TO analyzing_user;
```

In a big company, you often don't even want to give SELECT access to most users, because there might be private data in a table, like a user's email address or name. Many companies have anonymized versions of their databases that they can query on without worrying about access to private information.