CS50's Introduction to Databases with SQL

OpenCourseWare

Donate ↗ (https://cs50.harvard.edu/donate)

Carter Zenke (https://carterzenke.me)
carter@cs50.harvard.edu
 (https://github.com/carterzenke) (https://www.linkedin.com/in/carterzenke/)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
 (https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan) (https://www.threads.net/@davidjmalan)
(https://twitter.com/davidjmalan)

Lecture 3

Introduction
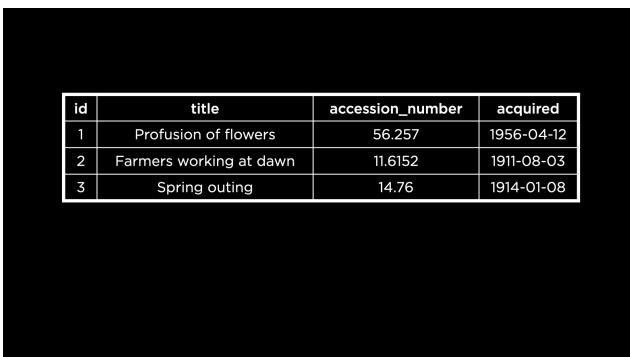
- Last week, we learned how to create our own database schema. In this lecture, we'll explore how to add, update, and delete data in our databases.

- The Boston MFA (Museum of Fine Arts) is a century-old museum in Boston. The MFA manages a vast collection of historical and contemporary artifacts and artwork. They likely use a database of some kind to store data about their art and artifacts.

- When a new artifact is added to their collection, we can imagine they would insert the corresponding data to their database. Similarly, there are use cases in which data might need to be read, updated or deleted.

- We will focus now on the creation (or insertion) of data in a Boston MFA database.

Database Schema

- Consider this schema that the MFA might use for its collection.

id	title	accession_number	acquired
1	Profusion of flowers	56.257	1956-04-12
2	Farmers working at dawn	11.6152	1911-08-03
3	Spring outing	14.76	1914-01-08

- Each row of data contains the title for a piece of artwork along with the `accession_number` which is a unique ID used by the museum internally. There is, too, a date indicating when the art was acquired.

- The table contains an ID which serves as the primary key.

- We can imagine that the database administrator of the MFA runs an SQL query to insert each of these pieces of artwork into the table.

- To understand how this works, let us first create a database called `mfa.db`. Next, we read the schema file `schema.sql` into the database. This schema file, already given to us, helps us create the table `collections`.

- To confirm that the table has been created, we can select from the table.

```
SELECT * FROM "collections";
```

This should give us an empty result, because the table doesn't have any data yet.

Inserting Data

- The SQL statement `INSERT INTO` is used to insert a row of data into a given table.

```
INSERT INTO "collections" ("id", "title", "accession_number", "acquir
VALUES (1, 'Profusion of flowers', '56.257', '1956-04-12');
```

We can see that this command requires the list of columns in the table that will receive new data and the values to be added to each column, in the same order.

- Running the `INSERT INTO` command returns nothing, but we can run a query to confirm that the row is now present in `collections`.

```sql
SELECT * FROM "collections";
```

- We can add more rows to the database by inserting multiple times. However, typing out the value of the primary key manually (as 1, 2, 3 etc.) might result in errors. Thankfully, SQLite can fill out the primary key values automatically. To make use of this functionality, we omit the ID column altogether while inserting a row.

```sql
INSERT INTO "collections" ("title", "accession_number", "acquired")
VALUES ('Farmers working at dawn', '11.6152', '1911-08-03');
```

We can check that this row has been inserted with an `id` of 2 by running

```sql
SELECT * FROM "collections";
```

Notice that the way SQLite fills out the primary key values is by incrementing the previous primary key—in this case, 1.

Questions

> If we delete a row with the primary key 1, will SQLite automatically assign a primary key of 1 to the next inserted row?

- No, SQLite actually selects the highest primary key value in the table and increments it to generate the next primary key value.

Other Constraints

- Opening the file `schema.sql` will pull up the schema for the database.

```sql
CREATE TABLE "collections" (
    "id" INTEGER,
    "title" TEXT NOT NULL,
    "accession_number" TEXT NOT NULL UNIQUE,
    "acquired" NUMERIC,
    PRIMARY KEY("id")
);
```

- It is specified that the accession number is unique. If we try to insert a row with a repeated accession number, we will trigger a error that looks like `Runtime error: UNIQUE constraint failed: collections.accession_number (19)`.

- This error informs us that the row we are trying to insert violates a constraint in the schema—specifically the `UNIQUE` constraint in this scenario.

- Similarly, we can try to add a row with a `NULL` title, violating the `NOT NULL` constraint.

```
INSERT INTO "collections" ("title", "accession_number", "acquired")
VALUES(NULL, NULL, '1900-01-10');
```

On running this, we will again see an error that looks like `Runtime error: NOT NULL constraint failed: collections.title (19)`.

- In this manner, the schema constraints are guardrails that protect us from adding rows that do not follow the schema of our database.

Inserting Multiple Rows

- We may need to insert more than one row at a time while writing into a database. One way to do this is to separate out the rows using commas in the `INSERT INTO` command.

```
INSERT INTO table (column0, …)
VALUES
(value0, …),
(value1, …),
…;
```

Inserting multiple rows at once in this manner allows the programmer some convenience. It is also a faster, more efficient way of inserting rows into a database.

- Let us now insert two new paintings into the `collections` table.

```
INSERT INTO "collections" ("title", "accession_number", "acquired")
VALUES
('Imaginative landscape', '56.496', NULL),
('Peonies and butterfly', '06.1899', '1906-01-01');
```

The museum may not always know exactly when a painting was acquired, hence it is possible for the `acquired` value to be `NULL`, as is the case for the first painting we just inserted.

- To see the updated table, we can select all rows from the table as always.

```
SELECT * FROM "collections";
```

- Our data could also be stored in a [comma-separated values (https://en.wikipedia.org/wiki/Comma-separated_values)](https://en.wikipedia.org/wiki/Comma-separated_values) format, or CSV. Observe in the following example how the values in each row are separated by a comma.

```
id,title,accession_number,acquired
1,Profusion of flowers,56.257,1956-04-12
2,Farmers working at dawn,11.6152,1911-08-03
3,Spring outing,14.76,1914-01-08
4,Imaginative landscape,56.496,
5,Peonies and butterfly,06.1899,1906-01-01
```

- SQLite makes it possible to import a CSV file directly into our database. To do this, we need to start from scratch. Let us leave this database `mfa.db` and then remove it.

- We already have a CSV file called `mfa.csv` that contains the data we need. On opening up this file, we can note that the first row contains the column names, which match exactly with the column names of our table `collections` as per the schema.

- First, let us create again the database `mfa.db` and read the schema file as we did earlier.

- Next, we can import the CSV by running a SQLite command.

```
.import --csv --skip 1 mfa.csv collections
```

  The first argument, `--csv` indicates to SQLite that we are importing a CSV file. This will help SQLite parse the file correctly. The second argument indicates that the first row of the CSV file (the header row) needs to be skipped, or not inserted into the table.

- We can select all the data from the `collections` table to see that every painting from `mfa.csv` has been successfully imported into the table.

- The CSV file we just inserted contained primary key values (1, 2, 3 etc.) for each row of data. However, it is more likely that CSV files we work with will not contain the ID or primary key values. How can we have SQLite insert them automatically?

- To try this out, let's open up `mfa.csv` in our codespace and delete the `id` column from the header row, along with the values in each column. This is what `mfa.csv` should look like once we finish editing:

```
title,accession_number,acquired
Profusion of flowers,56.257,1956-04-12
Farmers working at dawn,11.6152,1911-08-03
Spring outing,14.76,1914-01-08
Imaginative landscape,56.496,
Peonies and butterfly,06.1899,1906-01-01
```

- We will also delete all the rows that are already within the `collections` table.

```
DELETE FROM "collections";
```

- Now, we want to import this CSV file into a table. However, the collections table (as per our schema) must have four columns in every row. This new CSV file contains only three columns for every row. Hence, we cannot proceed to import in the same way we did before.

- To successfully import the CSV file without ID values, we will to use a temporary table:

```
.import --csv mfa.csv temp
```

Notice how we don't use the argument `--skip 1` with this command. This is because SQLite is capable of recognizing the very first row of CSV data as the header row, and converts those into the column names of the new temp table.

- We can see the data within the temp table by querying it.

```
SELECT * FROM "temp";
```

- Next, we will select the data (without primary keys) from temp and move it to collections, which was the goal all along! We can use the following command to achieve this.

```
INSERT INTO "collections" ("title", "accession_number", "acquired")
SELECT "title", "accession_number", "acquired" FROM "temp";
```

In this process, SQLite will automatically add the primary key values in the id column.

- Just to clean up our database, we can also drop the temp table once we're done moving data.

```
DROP TABLE "temp";
```

Questions

Can we place columns in specific positions while inserting into a table?

- While we can change the ordering of values in the INSERT INTO command, we usually can't change the ordering of the column names themselves. The order of column names follows the same order used while creating the table.

What happens if one of the multiple rows we are trying to insert violates a table constraint?

- While trying to insert multiple rows into a table, if even one of them violates a constraint, the insertion command will result in an error and none of the rows will be

inserted!

> After inserting data from the CSV, one of the cells was empty and not `NULL`. Why did this happen?

- When we imported data from the CSV file, one of the `acquired` values was missing! This was interpreted as text and hence, read into the table as an empty text value. We can run queries on the table after importing to convert these empty values into `NULL` if required.

Deleting Data

- We saw previously that running the following command deleted all rows from the table `collections`. (We don't want to actually run this command now or we'll lose all the data in the table!)

```
DELETE FROM "collections";
```

- We can also delete rows that match specific conditions. For example, to delete the painting "Spring outing" from our table `collections` we can run:

```
DELETE FROM "collections"
WHERE "title" = 'Spring outing';
```

- To delete any paintings with the date acquired as `NULL` we can run

```
DELETE FROM "collections"
WHERE "acquired" IS NULL;
```

- As we always do, we will make sure the deletion worked as expected by selecting all data from the table.

```
SELECT * FROM "collections";
```

We see that the "Spring outing" and "Imaginative landscape" paintings are not in the table anymore.

- To delete rows pertaining to paintings older than 1909, we can run
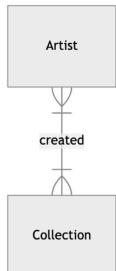
```
DELETE FROM "collections"
WHERE "acquired" < '1909-01-01';
```

Using the `<` operator here, we are finding the paintings acquired **before** January 1, 1909. These are the paintings that will be deleted on running the query.

- There might be cases where deleting some data could impact the integrity of a database. Foreign key constraints are a good example. A foreign key column references

the primary key of a different table. If we were to delete the primary key, the foreign key column would have nothing to reference!

- Consider now an updated schema for the MFA database, containing information not just about artwork but also artists. The two entities Artist and Collection have a many-to-many relationship—a painting can be created by many artists and a single artist can also create many pieces of artwork.

- Here is a database implementing the above ER Diagram.

The `artists` and `collections` tables have primary keys—the ID columns. The `created` table references these IDs in its two foreign key columns.

- Given this database, if we choose to delete the unidentified artist (with the ID 3), what would happen to the rows in the table `created` with an `artist_id` of 3? Let's try it out.

- After opening up `mfa.db`, we can now see the updated schema by running the `.schema` command. The `created` table does indeed have two foreign key constraints, one for the artist ID and one for the collection ID.

- Now, we can try to delete from the `artists` table.

```
DELETE FROM "artists"
WHERE "name" = 'Unidentified artist';
```

On running this, we get an error very similar to ones we have seen before in this class: `Runtime error: FOREIGN KEY constraint failed (19)`. This error notifies us that deleting this data would violate the foreign key constraint set up in the `created` table.

- How do we ensure that the constraint is not violated? One possibility is to delete the

corresponding rows from the `created` table before deleting from the `artists` table.

```
DELETE FROM "created"
WHERE "artist_id" = (
    SELECT "id"
    FROM "artists"
    WHERE "name" = 'Unidentified artist'
);
```

This query effectively deletes the artist's *affiliation* with their work. Once the affiliation no longer exists, we can delete the artist's data without violating the foreign key constraint. To do this, we can run

```
DELETE FROM "artists"
WHERE "name" = 'Unidentified artist';
```

- In another possibility, we can specify the action to be taken when an ID referenced by a foreign key is deleted. To do this, we use the keyword `ON DELETE` followed by the action to be taken.

    - `ON DELETE RESTRICT` : This restricts us from deleting IDs when the foreign key constraint is violated.

    - `ON DELETE NO ACTION` : This allows the deletion of IDs that are referenced by a foreign key and nothing happens.

    - `ON DELETE SET NULL` : This allows the deletion of IDs that are referenced by a foreign key and sets the foreign key references to `NULL` .

    - `ON DELETE SET DEFAULT` : This does the same as the previous, but allows us to set a default value instead of `NULL` .

    - `ON DELETE CASCADE` : This allows the deletion of IDs that are referenced by a foreign key and also proceeds to cascadingly delete the referencing foreign key rows. For example, if we used this to delete an artist ID, all the artist's affiliations with the artwork would also be deleted from the `created` table.

- The latest version of the schema file implements the above method. The foreign key constraints now look like

```
FOREIGN KEY("artist_id") REFERENCES "artists"("id") ON DELETE CASCADE
FOREIGN KEY("collection_id") REFERENCES "collections"("id") ON DELETE
```

Now running the following `DELETE` statement will not result in an error, and will cascade the deletion from the `artists` table to the `created` table:

```
DELETE FROM "artists"
WHERE "name" = 'Unidentified artist';
```

To check that this cascading deletion worked, we can query the `created` table:

```
SELECT * FROM "created";
```

We observe that none of the rows have an ID of 3 (the ID of the artist deleted from the `artists` table).

Questions

> We just deleted an artist with the ID of 3. Is there any way to make the next inserted row have an ID of 3?

- By default, as we discussed before, SQLite will select the largest ID present in the table and increment it to obtain the next ID. But we can use the `AUTOINCREMENT` keyword while creating a column to indicate that any deleted ID should be repurposed for a new row being inserted into the table.

Updating Data

- We can easily imagine scenarios in which data in a database would need to be updated. Perhaps, in the case of the MFA database, we find out that the painting "Farmers working at dawn" originally mapped to an "Unidentified artist" was actually created by the artist Li Yin.

- We can use the update command to make changes to say, the affiliation of a painting. Here is the syntax of the update command.

```
UPDATE table
SET column0 = value0, …
WHERE condition;
```

- Let's change this affiliation for "Farmers working at dawn" in the `created` table using the above syntax.

```sql
UPDATE "created"
SET "artist_id" = (
    SELECT "id"
    FROM "artists"
    WHERE "name" = 'Li Yin'
)
WHERE "collection_id" = (
    SELECT "id"
    FROM "collections"
    WHERE "title" = 'Farmers working at dawn'
```

```
    );
```

The first part of this query specifies the table to be updated. The next part retrieves the ID of Li Yin to set as the new ID. The last part selects the row(s) in `created` which will be updated with the ID of Li Yin, which is the painting "Farmers working at dawn"!

Fin

- This brings us to the conclusion of Lecture 3 about Writing in SQL!