Introduction to SQL Unit 7

NYC Data Science Academy

Outline

- Data Manipulation
  - `CREATE TABLE`
  - `INSERT`
  - `UPDATE`
  - `DELETE`
  - `TRUNCATE`
  - `DROP`
- Loading data from files
  - `LOAD`
  - Delimiters
- Data Integrity
  - `PRIMARY KEY`
  - `FOREIGN KEY`

Outline

- **Data Manipulation**
  - `CREATE TABLE`
  - `INSERT`
  - `UPDATE`
  - `DELETE`
  - `TRUNCATE`
  - `DROP`
- Loading data from files
  - `LOAD`
  - Delimiters
- Data Integrity
  - `PRIMARY KEY`
  - `FOREIGN KEY`

Data Manipulation

So far we have been querying data, without changing them in the database. SQL also allows manipulating the data with some simple statements.

CREATE TABLE

A table can be created with the command `CREATE TABLE`. The syntax is as below:

```
CREATE TABLE table_name (
  column_name column_type [options],
  ...
  )
```

CREATE TABLE

An example would be:

```
CREATE TABLE actors (
    biography VARCHAR(255),
    birthday DATE DEFAULT NULL,
    deathday DATE DEFAULT NULL,
    gender INT DEFAULT NULL,
    name VARCHAR(255),
    place_of_birth VARCHAR(255),
    facebook_likes INT DEFAULT NULL
);
```

- The `table_name` in this case is `actors`
- The first column is named by `biography`, whose type is `VARCHAR`. Notice that there is no options specified.
- The second column is named by `birthday`, whose type is `DATE`. We do specify the default value to be `NULL`.
  More options can be found in the MySQL reference page.

CREATE TABLE

To confirm the desired table is created, we may:

```
DESCRIBE actors;
```

or:

```
SHOW COLUMNS FROM actors;
```

INSERT

A table can be created only with the schema, as we specified in the last slide. The values can be empty as you would see if you query the `actors` table now.

To actually insert some value, we may use:

```
INSERT INTO table_name
(column_name_1, column_name_2, ...)
VALUES
(value_row1_col1, value_row1_col2, ...),
(value_row2_col1, value_row2_col2, ...),
...
;
```

INSERT

An example would be:

```
INSERT INTO actors
(birthday,deathday,gender,name,place_of_birth,facebook_likes)
VALUES
("1942-08-17",NULL,0,"Roshan Seth","Patna, Bihar, India",61.0);
```

Now you have the first row in you table. Try the query below to see the data inserted:

```
SELECT * FROM actors;
```

UPDATE

Some information might need to be updated. This can be done by:

```
UPDATE table_name SET column_name = new_value
[WHERE where_condition];
```

Notice that the `WHERE` clause is necessary to identify the row for updating.

UPDATE

An example would be:

```
UPDATE actors SET facebook_likes = 68
WHERE name = "Roshan Seth";
```

It is likely that the number of 'like' in Facebook would need to be updated. This applies to the certain actor, so we need to identify him by the `name`.

DELETE

Beside updating the records, we often would like to remove some. The command `DELETE` can help:

```
DELETE FROM tbl_name
[WHERE where_condition];
```

DELETE

An example is like:

```
DELETE FROM actors WHERE name = "Roshan Seth";
```

Try to query the table `actors` to see the record is removed.

Exercise

1. Insert the data from `first_exercise.csv` into the table `actors`.
2. Delete the record of `Actor One`.

Solution

1.

```
INSERT INTO actors (biography,birthday,deathday,gender,
name,place_of_birth,facebook_likes)
VALUES
("Bio for Actor One","1950-01-11",NULL,0,
"Actor One","New York, New York, USA",100),
("Bio for Actor Two,
with multiple lines","1915-01-01","2015-01-01",0,
"Actor Two","New York, New York, USA",1000);
```

2.

```
DELETE FROM actors WHERE name = "Actor One";
```

TRUNCATE

To empty a table, we may use `TRUNCATE`. The syntax is very simple:

```
TRUNCATE TABLE table_name;
```

However, one need to be very careful when using `TRUNCATE` since it removes everything in a table.

TRUNCATE

An example would be:

```
TRUNCATE TABLE actors;
```

Try the command now:

```
SELECT * FROM actors;
```

No record would be returned.

TRUNCATE

Also, we may still inspect the columns in the table `actors`:

```
DESCRIBE actors;
```

We can see that the schema still exists. This is not the case for the next command that we will introduce.

DROP

To fully remove a table, without even holding the schema, we may use `DROP`. The syntax is very simple:

```
DROP TABLE table_name;
```

Again, one need to be very careful when using `DROP`.

DROP

To try `DROP`, we reconstruct the table:

```
INSERT INTO actors (biography,birthday,deathday,gender,
    name,place_of_birth,facebook_likes)
VALUES
("", "1942-08-17",NULL,0,"Roshan Seth",
    "Patna, Bihar, India",61.0),
("Bio for Actor One","1950-01-11",NULL,0,"Actor One",
    "New York, New York, USA",100),
("Bio for Actor Two,
with multiple lines","1915-01-01","2015-01-01",0,"Actor Two",
    "New York, New York, USA",1000);
```

Then we may inspect the table by:

```
SELECT * FROM actors;
```

Exercise

1. `DROP` the whole `actors` table.
2. Inspect the `actors` table.
3. Inspect the schema of the `actors` table.
4. What do you think the difference is between `TRUNCATE` and `DROP`?

Solution

1.

```
DROP TABLE actors;
```

2.

```
SELECT * FROM actors;
```

3.

```
DESCRIBE actors;
```

4. The whole table, including the schema, is removed by `DROP`.

Outline

- Data Manipulation
  - `CREATE TABLE`
  - `INSERT`
  - `UPDATE`
  - `DELETE`
  - `TRUNCATE`
  - `DROP`
- **Loading data from files**
  - **`LOAD`**
  - **Delimiters**
- Data Integrity
  - `PRIMARY KEY`
  - `FOREIGN KEY`

LOAD

So far we have seen how we can create a schema (an empty table) with the `CREATE TABLE` command, and how we can `INSERT` values to the table. Some times we would like to be able to load the data from files.

Most commonly used syntax would be like:

```
LOAD DATA LOCAL INFILE file_name
INTO TABLE table_name
FIELDS TERMINATED BY string_1 ENCLOSED BY string_2
LINES TERMINATED BY string_3
IGNORE int LINES
;
```

LOAD

Below we explain each component in the command above:

1. `LOAD DATA LOCAL` allow import data from a local file.
2. `INFILE` specifies the path to the file.
3. `INTO TABLE` specifies the name of the table where we load the data into.
4. `FIELDS TERMINATED BY` specifies the delimiters. The delimeters enclosed by the string specified in `ENCLOSED BY` would not terminate a field. We will explain in greater details for these two terms.
5. A line would be terminated by the string specified in `LINE TERMINATED BY`. A line represents a single record in the data.
6. As many lines as specified in `IGNORE LINES` would be ignored. This often prevents treating the column names as part of the data.

Delimiters

Most tables come with multiple columns. Within each row, fields are often separated by the **delimeters**. The most commonly used file format to save data are:

- CSV: comma-separated-values files

or

- TSV: tab-separated-values files

The difference is the delimiters that separate values.

Delimiters

For example, to represent the table

A	B	C
1	2	3
4	5	6

a csv file would be like:

```
A,B,C
1,2,3
4,5,6
```

and a tsv file would be like:

```
A	B	C
1	2	3
4	5	6
```

For different files, we need to choose an appropriate string for `FIELDS TERMINATED BY`.

Delimiters

Some times, we would like to escape some delimeters, for example:

name	place_of_birth
Roshan Seth	Patna, Bihar, India
Alona Tal	Herzlia - Israel
Horatio Sanz	Santiago, Chile
Jake Wood	Westminster, London, England, UK
Seth Gilliam	New York - USA

Delimiters

In the column `place_of_birth`, we have the value `Patna, Bihar, India`, which we definitely don't want to split. Very often, the data would be saved as:

```
name, place_of_birth
"Roshan Seth","Patna, Bihar, India"
"Alona Tal","Herzlia - Israel"
"Horatio Sanz","Santiago, Chile"
"Jake Wood","Westminster, London, England, UK"
"Seth Gilliam","New York - USA"
```

Specifying `ENCLOSED BY = '"'` will ignore the comma enclosed by ". Therefore, `Patna, Bihar, India` would be taken for the value of one cell, instead of three.

LOAD

An example would be:

```
LOAD DATA LOCAL INFILE '<your path to>/actors_bf_2016.csv'
INTO TABLE actors
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
;
```

Notice that in sql command, strings have to be enclosed by quotes. Therefore,

- `'./actors_bf_2016.csv'` indicates `./actors_bf_2016.csv`
- `','` indicates `,`
- `'"'` indicates `"`
- `'\n'`indicates `\n`

Outline

- Data Manipulation
  - `CREATE TABLE`
  - `INSERT`
  - `UPDATE`
  - `DELETE`
  - `TRUNCATE`
  - `DROP`
- Loading data from files
  - `LOAD`
  - Delimiters
- **Data Integrity**
  - **`PRIMARY KEY`**
  - **`FOREIGN KEY`**

Data Integrity

It is very convenient to load data from files. However, importing multiple records into the database can cause problem for data integrity. For example, if we load the data from `actors_bf_2016.csv`:

```
LOAD DATA LOCAL INFILE '<your path to>/actors_bf_2016.csv'
INTO TABLE actors
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
;
```

we can check

```
SELECT name, COUNT(name) FROM actors
GROUP BY name ORDER BY COUNT(name) DESC;
```

PRIMARY KEY

To avoid this issue, it would be better to have constraint at the first place. Let's drop the table:

```
DROP TABLE actors;
```

Create the schema again, but with `PRIMARY KEY` this time:

```
CREATE TABLE actors (
    biography varchar(255),
    birthday date DEFAULT NULL,
    deathday date DEFAULT NULL,
    gender int DEFAULT NULL,
    name varchar(255),
    place_of_birth varchar(255),
    facebook_likes int DEFAULT NULL,

    PRIMARY KEY (name)
);
```

PRIMARY KEY

Then we load the data again and there is no duplicate this time:

```
LOAD DATA LOCAL INFILE '<your path to>/actors_bf_2016.csv'
INTO TABLE actors
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
;

SELECT name, COUNT(name)
FROM actors
GROUP BY name
ORDER BY COUNT(name) DESC;
```

PRIMARY KEY

NOTE:

We demonstrate the functionality of `PRIMARY KEY` with the column `name`. However, it is very uncommon that `name` is used as the `PRIMARY KEY`, since it is possible that two different actors are having the same name.

Deciding that the repeating names are actually from different actors or are duplicate by mistake is part of **data cleaning**. In general, data cleaning is very time consuming. Simply adding a primary key is unlikely to solve the problem.

Exercise

Create a proper schema with `movie_title` as the primary key, and load the data from `'movies_bf_2016.csv'`. Make a proper query to see that there is no duplicate. To avoid excessive typing, we have the schema ready for copy and paste:

```
color varchar(255), director_name varchar(255),
num_critic_for_reviews int DEFAULT NULL,
duration int DEFAULT NULL,
actor_2_name varchar(255), gross int DEFAULT NULL,
genres varchar(255), actor_1_name varchar(255),
movie_title varchar(255),
num_voted_users int DEFAULT NULL,
cast_total_facebook_likes int DEFAULT NULL,
actor_3_name varchar(255),
facenumber_in_poster int DEFAULT NULL,
plot_keywords varchar(255), movie_imdb_link varchar(255),
num_user_for_reviews int DEFAULT NULL,
language varchar(255), country varchar(255),
content_rating varchar(255),
budget int DEFAULT NULL, title_year int DEFAULT NULL,
imdb_score float DEFAULT NULL, aspect_ratio float DEFAULT NULL,
movie_facebook_likes int DEFAULT NULL,
```

Solution

- Step 1

```
CREATE TABLE movies (
    color varchar(255), director_name varchar(255),
    num_critic_for_reviews int DEFAULT NULL,
    duration int DEFAULT NULL,
    actor_2_name varchar(255), gross int DEFAULT NULL,
    genres varchar(255), actor_1_name varchar(255),
    movie_title varchar(255),
    num_voted_users int DEFAULT NULL,
    cast_total_facebook_likes int DEFAULT NULL,
    actor_3_name varchar(255),
    facenumber_in_poster int DEFAULT NULL,
    plot_keywords varchar(255), movie_imdb_link varchar(255),
    num_user_for_reviews int DEFAULT NULL,
    language varchar(255), country varchar(255),
    content_rating varchar(255),
    budget int DEFAULT NULL, title_year int DEFAULT NULL,
    imdb_score float DEFAULT NULL, aspect_ratio float DEFAULT NULL,
    movie_facebook_likes int DEFAULT NULL,

    PRIMARY KEY (movie_title)
);
```

Solution

- Step 2

```
LOAD DATA LOCAL INFILE '<your path to>/movies_bf_2016.csv'
INTO TABLE movies
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
;
```

- Step 3

```
SELECT movie_title, COUNT(movie_title)
FROM movies
GROUP BY movie_title
ORDER BY COUNT(movie_title) DESC;
```

Foreign Key

It can be hard to manage the data integrity when they spread over multiple tables.

Consider our example of `movies` table, one of whose columns is the name of the leading actor, whose detail information is recorded in another table, `actors`.

What if we would like to remove an actor from the `actors` table?

We might need to know the movies in our `movies` table that are cast by this actor. We might want to replace this name by `NULL` in `movies` or to just prevent the removing.

Apparently we don't want to manully find the rows with this actor; or replace their name by `NULL`. An easier way to do this is to use `FOREIGN KEY`.

Foreign Key

The syntax below add the foreign key to a existing table:

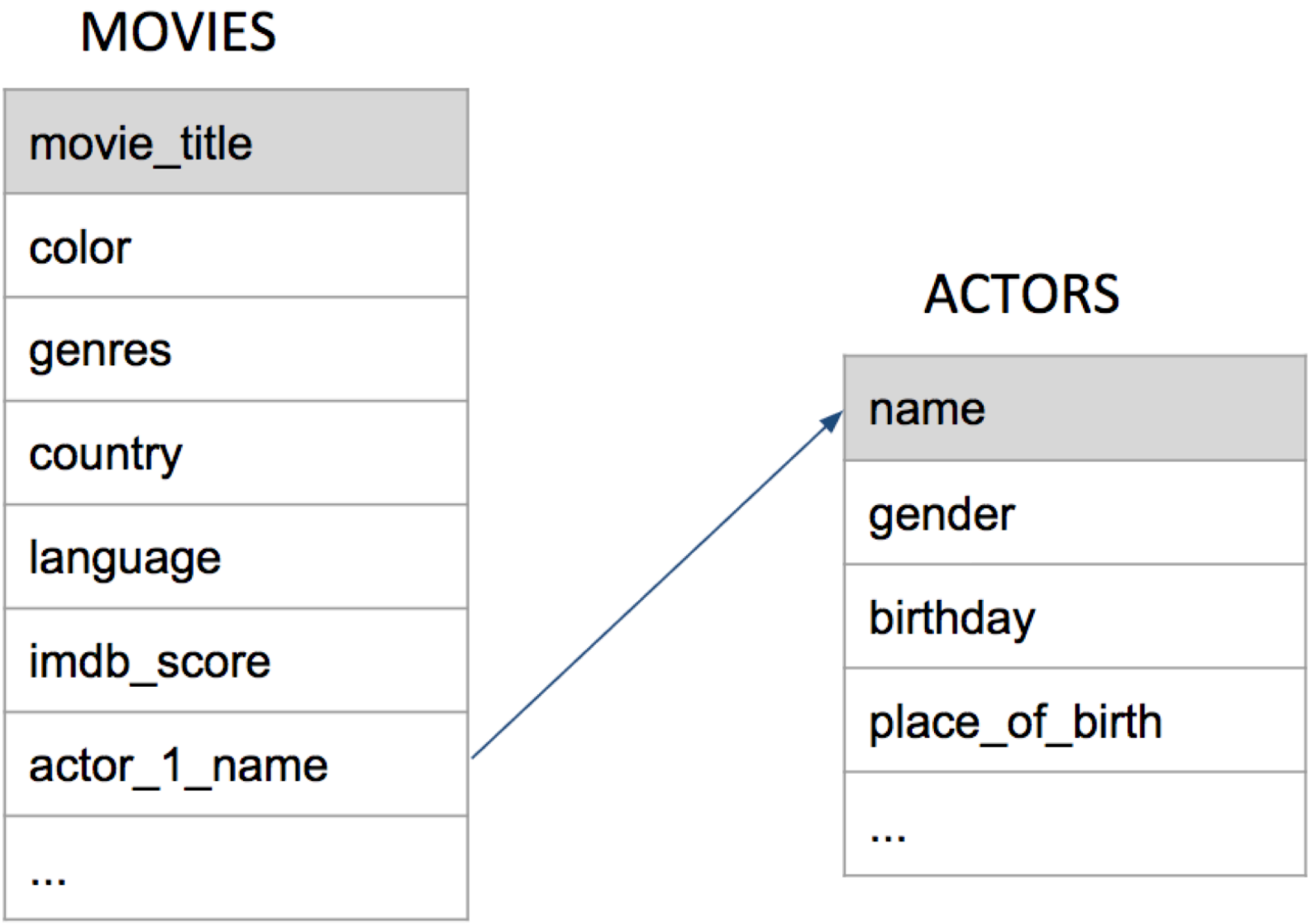
```
ALTER TABLE child_table
ADD CONSTRAINT constraint_name
FOREIGN KEY index_col_name
REFERENCES parent_table(index_col_name);
```

To expain each part in the code above, let's considet an example:

```
ALTER TABLE movies
ADD CONSTRAINT FK_ActorMovie
FOREIGN KEY (actor_1_name) REFERENCES actors(name);
```

(Visualization in the next slide)

Foreign Key


MOVIES
movie_title
color
genres
country
language
imdb_score
actor_1_name
...
ACTORS
name
gender
birthday
place_of_birth
...

Foreign Key

From the command above:

- `ALTER TABLE movies` indicates that we are working on the table `movies`.
- `ADD CONSTRAINT` means we are imposing some constraint, `FOREIGN KEY` in this case, to the table. The foreign keys are a special case of **constraint**. It is indeed a constraint since some operations would be prohibited with foreign keys.
- `FK_ActorMovie` is the name that we assign to the constaint. An error or warning message would be thrown once we attemp some operations that violate the constraint. A reasonable constraint name is of great help for figuring out the violation.
- `FOREIGN KEY (actor_1_name) REFERENCES actors(name)` links the column `actor_1_name` in `movies` (the table we are altering) to the column `name` in `actors`.

Exercise

Does the code

```
ALTER TABLE movies
ADD CONSTRAINT FK_ActorMovie
FOREIGN KEY (actor_1_name) REFERENCES actors(name);
```

work? Why?

Solution

It fails with a error message similar to below:

```
ERROR 1452 (23000): Cannot add or update a child row:
a foreign key constraint fails
```

because foreign key doesn't allow `movies` to have an actor that is not in `actors`.

```
SELECT COUNT(*) FROM movies WHERE actor_1_name NOT IN
(SELECT name FROM actors);
```

Foreign Key

Let's drop the `movies` and reconstruct it with a foreign key:

```
DROP TABLE movies;
```

Foreign Key

```
CREATE TABLE movies (
    color varchar(255), director_name varchar(255),
    num_critic_for_reviews int DEFAULT NULL,
    duration int DEFAULT NULL, actor_2_name varchar(255),
    gross int DEFAULT NULL, genres varchar(255),
    actor_1_name varchar(255), movie_title varchar(255),
    num_voted_users int DEFAULT NULL,
    cast_total_facebook_likes int DEFAULT NULL,
    actor_3_name varchar(255),
    facenumber_in_poster int DEFAULT NULL,
    plot_keywords varchar(255), movie_imdb_link varchar(255),
    num_user_for_reviews int DEFAULT NULL,
    language varchar(255), country varchar(255),
    content_rating varchar(255), budget int DEFAULT NULL,
    title_year int DEFAULT NULL,
    imdb_score float DEFAULT NULL,
    aspect_ratio float DEFAULT NULL,
    movie_facebook_likes int DEFAULT NULL,

  PRIMARY KEY (movie_title),
  CONSTRAINT FK_actor_1_moviesactors
  FOREIGN KEY (actor_1_name) REFERENCES actors(name)
);
```

Foreign Key

Notice that it's possible to add the foreign key without specifying the constraint name, as in the code below:

```
CREATE TABLE movies (
    ...

    PRIMARY KEY (movie_title),
    FOREIGN KEY (actor_1_name) REFERENCES actors(name)
);
```

However, when we need to refer to this particular foreign key (we will, soon), it's much more convenient if we have a good name for the foreign key. In fact, it is recommended to always name the foreign key properly when creating one.

Foreign Key

```
LOAD DATA LOCAL INFILE '<path to your>/movies_bf_2016.csv'
INTO TABLE movies
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
;
```

Now every `actor_1_name` is in the `name` column in `actors`:

```
SELECT COUNT(*) FROM movies WHERE actor_1_name NOT IN
(SELECT name FROM actors);
```

Foreign Key

Every actor in `actor_1_name` of `movies` needs to be in the `name` in `actors`. This also said, deleting is also restricted. For example,

```
SELECT movie_title, actor_1_name FROM movies limit 1;
```

Try to delete the actor from `actors`, what would happen?

Foreign Key

If we want to relax the constraint a little bit: we still don't allow `movies` to have actor that is not in `actors`, but we allow deleting by replacing the `actor_1_name` by `NULL`.

To alter the constraint, we may remove the old one and then add the desired one. Below is the command to remove old constraint, where we see again why we would like to have a good key name:

```
ALTER TABLE movies
DROP FOREIGN KEY FK_actor_1_moviesactors;
```

Foreign Key

We modify the constraint with:

```
ALTER TABLE movies
ADD CONSTRAINT FK_actor_1_moviesactors
FOREIGN KEY (actor_1_name) REFERENCES actors(name)
    ON DELETE SET NULL;
```

The main difference is that we add

```
ON DELETE SET NULL
```

This indicates that upon deleting in child table, the corresponding foreign key would be replaced by `NULL`.

Foreign Key

We can then successfully delete the actor now:

```
DELETE FROM actors WHERE name = '50 Cent';

SELECT movie_title, actor_1_name FROM movies limit 1;
```

Exercise

- Upload movies_af_2016.csv and actors_af_2016.csv
- How many new movies were added?

Solution

```
LOAD DATA LOCAL INFILE '<your path to>/actors_af_2016.csv'
INTO TABLE actors
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
;
LOAD DATA LOCAL INFILE '<your path to>/movies_af_2016.csv'
INTO TABLE movies
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
;
```

Multiple Foreign Key

Multiple foreign keys is also possible. We will also demonstrate different ways of managing integrity with the code.

Let's drop the current table:

```
DROP TABLE movies;
```

CASCADE

Create the schema with the same command as previous syntax, except replacing the constraint part by:

```
CREATE TABLE movies (
  ...

    PRIMARY KEY (movie_title),
    CONSTRAINT FK_actor_1_moviesactors
    FOREIGN KEY (actor_1_name)
      REFERENCES actors(name)
        ON UPDATE CASCADE
        ON DELETE SET NULL,
    FOREIGN KEY (actor_2_name)
      REFERENCES actors(name)
        ON UPDATE CASCADE
        ON DELETE SET NULL,
    FOREIGN KEY (actor_3_name)
      REFERENCES actors(name)
        ON DELETE CASCADE
);
```

CASCADE

Notice that we have `ON DELETE` defining the effect on the corresponding rows in `movies` when **deleting** rows in `actors`, it could be either `CASCADE` or `SET NULL` (We will explain what they mean with examples). Here we see that when deleting rows from `actors`,

- the movie with those actors as the `actor_1_name` or `actor_2_name`, the effect would be `SET NULL`.
- the movie with those actors as the `actor_3_name`, the effect would be `CASCADE`.

On the other hand, we have `ON UPDATE` defining the effect on the corresponding rows in `movies` when **updating** rows in `actors`, it could be either `CASCADE` or `SET NULL` (We will explain what they mean with examples). Here we see that when deleting rows from `actors`,

- the movie with those actors as the `actor_1_name` or `actor_2_name`, the effect would be `CASCADE`.
- if there exist movies with those actors as the `actor_3_name`, the update would not be allowed.

CASCADE

Load the data first:

```
LOAD DATA LOCAL INFILE '<your path to>/movies_bf_2016.csv'
INTO TABLE movies
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
;
```

CASCADE

To understand the difference between `CASCADE` and `SET NULL`, we consider the row below:

```
SELECT movie_title, actor_1_name, actor_3_name FROM movies
WHERE actor_1_name = 'Jim Carter' OR actor_3_name = 'Jim Carter';
```

The treatment on delete for `actor_1_name` and `actor_2_name` are different as we can see from:

```
DELETE FROM actors WHERE name='Jim Carter';

SELECT movie_title, actor_1_name, actor_3_name FROM movies
WHERE
movie_title = "102 Dalmatians" OR
movie_title = "The Oxford Murders";
```

Notice that the deletion is broadcast to the whole row with `actor_3_name = "Jim Carter"`.

Updating

From the last query we got `actor_3_name = "Danny Sapani"`. Recall that we don't allow updating on `actor_3_name`, so

```
UPDATE actors SET name = 'changed!!' WHERE name = "Danny Sapani";
```

would return error message.

Updating

The other two actor columns allow updating, so we consider the following example:

```
SELECT movie_title, actor_1_name, actor_2_name, actor_3_name
FROM movies
WHERE
actor_1_name = "Ioan Gruffudd" OR
actor_2_name = "Ioan Gruffudd" OR
actor_3_name = "Ioan Gruffudd";
```

Notice that no row has `actor_3_name = "Ioan Gruffudd"`.

Updating

Updating `actors` whose `name = "Ioan Gruffudd"` works since `actor_3_name` is not involved.

```
UPDATE actors SET name = "changed!!"
WHERE name = "Ioan Gruffudd";
```

The changing would be broadcast to `actor_1_name` and `actor_2_name` as we will see from the query below:

```
SELECT movie_title, actor_1_name, actor_2_name, actor_3_name
FROM movies
WHERE actor_1_name = "changed!!"
OR actor_2_name = "changed!!"
OR actor_3_name = "changed!!";
```

Exercise

- Change "Chris Evans" to "Captain America".

Solution

```
UPDATE actors SET name = "Captain America"
WHERE name = "Chris Evans";
```