Primary and foreign keys - Text lecture
Primary Keys and Foreign Keys

While not always, most tables use Primary Keys to uniquely identify each row. In our exercises, we've had an auto incrementing integer that keeps the rows unique. Think about what would happen if you had lots of rows that were identical, how do you keep them separate or unique to identify them? Understanding keys is essential when we start exploring joining tables.
A primary key keeps every row unique. If you were going to join two tables together, you can use the primary key from one table to reference another In the examples in the video, we have 2 tables. Persons and Movies. Each has an integer as their primary key. Notice that the Persons Table uses the Primary key from Movies in the Favorite Movie column. This is an example of a foreign key. The Favorite Movie column contains they key of another table. Notice that rows in the Movies table can be referenced several times or not at all.

Another example in the video is another table 'MoviesPeopleHaveSeen'. Both PersonID and MovieID are foreign keys, but together they make the primary key of this table. This means, no two rows can have the exact same PersonID and MovieID combination. Setting up tables like this allows us to have details about the movie and details about the people recorded only once. If a detail about a move changes, we change it on the movies table in one place which saves space and simplifies updates, otherwise we would have to update every table that had movies in them.

Data prep for joins - Text lecture
Lets prepare some tables for Joins.
Generally, you don't want to rename tables, but let's expand the definition of our actors table and name it as people
RENAME TABLE actors TO people;

Now let's add some more people:

```
INSERT INTO people (FIRST_NAME , LAST_NAME)
VALUES ('Steven','Spielberg')
    ,('Shawn','Levy')
```

```
    ,('Jon','Turteltaub')
    ,('Rawson','Thurber')
    ,('John','Hamburg')
    ,('Adam','McKay')
    ,('Tom','McGrath')
    ,('David','Fincher');
```

Now let's add a director column to the movies table which will be a foreign key with the primary key of the people table:

```
ALTER TABLE movies
ADD COLUMN  Director INT NULL;
```

Now let's set directors for each row in the movies table:

```
UPDATE movies
SET Director = 1
WHERE MovieId IN (1,2,7);
UPDATE movies
SET Director = 17
WHERE MovieId IN (3,4,5);
UPDATE movies
SET Director = 18
WHERE MovieId = 6;
UPDATE movies
SET Director = 19
WHERE MovieId = 8;
UPDATE movies
SET Director = 20
WHERE MovieId = 9;
UPDATE movies
SET Director = 21
WHERE MovieId IN (10,11);
UPDATE movies
SET Director = 22
WHERE MovieId = 12;
UPDATE movies
SET Director = 23
WHERE MovieId = 13;
```

Let's create a characters table:

```
CREATE TABLE characters ( characterID INT NOT NULL AUTO_INCREMENT
                    ,MovieID INT NULL
                    ,ActorID INT NULL
                    ,CharacterName VARCHAR(100) NULL
                    ,PRIMARY KEY (characterID));
```

Let's add some characters to it with the foreign keys of MovieID and ActorID, along with Character Name in the movie:

```
INSERT INTO characters (MovieID , ActorID, CharacterName) VALUES
 (1 , 1 , 'Derek Zoolander')
,(1 , 2 , 'Hansel')
,(1 , 3 , 'Matilda Jeffries')
,(1 , 4 , 'Mugatu')
,(1 , 5 , 'Katinka')
,(1 , 6 , 'Maury Ballstein')
,(1 , 7 , 'J.P. Prewitt')
,(1 , 8 , 'Larry Zoolander')
,(1 , 9 , 'Todd')
,(2 , 1 , 'Derek Zoolander')
,(2 , 2 , 'Hansel')
,(2 , 3 , 'Matilda Jeffries')
,(2 , 4 , 'Mugatu')
,(2 , 5 , 'Katinka')
,(2 , 9 , 'Todd')
,(2 , 10, 'Alexanya Atoz')
,(2 , 11, 'Valentina Valencia')
,(2 , 12, 'Lenny Kravitz')
,(2 , 14, 'Justin Bieber')
,(2 , 15, 'Derek Jr.')
,(5 , 1 , 'Larry Daley')
,(5 , 2 , 'Jedediah')
,(4 , 1 , 'Larry Daley')
,(4 , 2 , 'Jedediah')
,(3 , 1 , 'Larry Daley')
,(3 , 2 , 'Jedediah')
,(3 , 1 , 'Laaa')
,(6 , 8 , 'Patrick Gates')
,(7 , 1 , 'Tugg Speedman')
,(7 , 3 , 'Rebecca')
```

```
,(7 , 8 , 'Jon Voight')
,(8 , 3 , 'Kate Veatch')
,(8 , 1 , 'White Goodman')
,(9 , 1 , 'Reuben Feffer');
```

Joins and aliases - Text lecture

Join:

```
SELECT *
FROM movies
    ,people
WHERE movies.DIRECTOR = people.ActorID ;
```

We simply list the tables with a comma to separate them and we define how the two tables relate to each other in the where clause.

When dealing with more than one table, we should always specify the table. Often times tables have columns with the same names and MySQL won't know which one you want.

Let's select some specific columns instead of all of them:

```
SELECT movies.TITLE
      ,movies.RELEASE_YEAR
      ,movies.RATING
      ,CONCAT(people.FIRST_NAME,' ',people.LAST_NAME) AS Director
FROM movies
    ,people
WHERE movies.DIRECTOR = people.ActorID ;
```

While specifying the table is necessary in many situations, and always a good habit, it causes a Lot of typing and can make the code look overwhelming. Which is why we use aliases

Aliases

```
SELECT A.TITLE
      ,A.RELEASE_YEAR
      ,A.RATING
      ,CONCAT(B.FIRST_NAME,' ',B.LAST_NAME) AS Director
FROM movies A
```

```
    ,people B
WHERE A.DIRECTOR = B.ActorID ;
```

You can use anything as the alias, but I have found that simply using from letters A-Z for each table is the best way. It's clean and straightforward. All of the examples and code going forward will incorporate this type of aliasing.

More about joins - Text lecture

Let's Join Movies with Characters

```
SELECT A.TITLE
      ,A.RELEASE_YEAR
      ,A.RATING
      ,B.CharacterName
      ,B.ActorID
FROM movies A
    ,characters B
WHERE A.MovieID = B.MovieID;
```

We can add additional tables:

```
SELECT A.TITLE
      ,A.RELEASE_YEAR
      ,A.RATING
      ,B.CharacterName
      ,CONCAT(C.FIRST_NAME,' ',C.LAST_NAME) AS Actor
FROM movies A
    ,characters B
    ,people C
WHERE A.MovieID = B.MovieID
 AND  B.ActorID  = C.ActorID;
```

We can reference the same table multiple times to pull information for various reasons. In the case below, we are using the People table to provide the name of both the directors and the actors

```
SELECT A.TITLE
      ,CONCAT(D.FIRST_NAME,' ',D.LAST_NAME) AS Director
      ,A.RELEASE_YEAR
      ,A.RATING
```

```
      ,B.CharacterName
      ,CONCAT(C.FIRST_NAME,' ',C.LAST_NAME) AS Actor
FROM movies A
    ,characters B
    ,people C
    ,people D
WHERE A.MovieID  = B.MovieID
 AND  B.ActorID  = C.ActorID
 AND  A.DIRECTOR = D.ActorID
 AND  A.TITLE LIKE 'Zoo%';
```

This can be written a different way. Listing tables in the FROM and joining them in the WHERE is perfectly acceptable, you can also join tables by putting everything in the FROM:

```
SELECT A.TITLE
      ,CONCAT(D.FIRST_NAME,' ',D.LAST_NAME) AS Director
      ,A.RELEASE_YEAR
      ,A.RATING
      ,B.CharacterName
      ,CONCAT(C.FIRST_NAME,' ',C.LAST_NAME) AS Actor
FROM movies A
       INNER JOIN characters B
         ON A.MovieID  = B.MovieID
       INNER JOIN people C
         ON B.ActorID  = C.ActorID
       INNER JOIN people D
         ON A.DIRECTOR = D.ActorID
WHERE A.TITLE LIKE 'Zoo%';
```

Both of these joins are Inner Joins. They mean that all of the rows in all of the tables need to exist. If no actors were listed for a movie, the movie won't show on the list. If no movies are associated with that actor, the actor won’t be returned.

Inner, Left, Outer and Right joins - Text lecture

Before we go any further with Joins, let's get the main types straight: INNER, LEFT OUTER, RIGHT OUTER and FULL OUTER.

There are a few ways to specify these types in SQL, but for the purposes of this course, we will stick to these terms: LEFT OUTER JOIN is often called a LEFT JOIN, and an INNER JOIN may be referred to as a JOIN. A FULL OUTER JOIN can be called an OUTER JOIN. etc.

Imagine we have two tables:

Table A
ColA
1
2
3
4
Table B
ColB
3
4
5
6
If we were to join these tables together, the number of rows returned would depend on the join we use. Notice 1 and 2 are unique to Table A, 5 and 6 are unique to table B. 3 and 4 are common to both tables.

```
SELECT *
FROM A INNER JOIN B ON A.ColA = B.ColB;
```
This would return:
```
ColA | ColB
 3   |  3
 4   |  4
```
INNER JOIN requires that the value be on both tables
```
SELECT *
FROM A LEFT OUTER JOIN B ON A.ColA = B.ColB;
```
This would return:
```
ColA | ColB
 1   | NULL
 2   | NULL
 3   |  3
 4   |  4
```

LEFT OUTER JOIN returns everything from the first table (left) whether or not the second table has a corresponding row
SELECT *
FROM A RIGHT OUTER JOIN B ON A.ColA = B.ColB;
This would return:
ColA | ColB
 3 | 3
 4 | 4
 NULL| 5
 NULL| 6
RIGHT OUTER JOIN returns everything from the second table (right) whether or not the first table has a corresponding row
SELECT *
FROM A FULL OUTER JOIN B ON A.ColA = B.ColB;
This would return:
ColA | ColB
 1 | NULL
 2 | NULL
 3 | 3
 4 | 4
 NULL| 5
 NULL| 6
FULL OUTER JOIN returns everything from both tables whether or not there are corresponding rows

Left outer joins - Text lecture
Let's say we want to report all of the actors and the movies that they directed:

```
SELECT A.FIRST_NAME
      ,A.LAST_NAME
      ,B.TITLE AS Directed
      ,B.RELEASE_YEAR
FROM people A
      INNER JOIN movies B
        ON B.DIRECTOR = A.ActorID;
```

If I want to have rows from the people table returned whether or not they directed a movie, I can use a left outer join:

```
SELECT A.FIRST_NAME
      ,A.LAST_NAME
      ,B.TITLE AS Directed
      ,B.RELEASE_YEAR
FROM people A
      LEFT OUTER JOIN movies B
        ON B.DIRECTOR = A.ActorID;
```

We can couple this concept with functions we discussed earlier like Min and Max

```
SELECT A.FIRST_NAME
      ,A.LAST_NAME
      ,MIN(C.RELEASE_YEAR) AS FirstDirected
      ,MAX(B.RELEASE_YEAR) AS LatestRelease
FROM people A
     LEFT OUTER JOIN movies B
      ON B.DIRECTOR = A.ActorID
     LEFT OUTER JOIN movies C
      ON C.DIRECTOR = A.ActorID
GROUP BY A.FIRST_NAME ,A.LAST_NAME
ORDER BY COUNT(B.MovieID) DESC;
```

When your joining tables, you can join them to the primary table or any table, but it is important to keep them in a logical order so that you can troubleshoot if you run into unexpected results.

```
SELECT A.FIRST_NAME
      ,A.LAST_NAME
      ,IFNULL(MIN(C.ReleaseYear),'N/A') AS FirstDirected
      ,IFNULL(MAX(B.ReleaseYear),'N/A') AS LatestRelease
FROM people A
      LEFT OUTER JOIN movies B
       ON B.director = A.ActorID
      LEFT OUTER JOIN movies C
       ON C.director = B.director
GROUP BY A.FIRST_NAME
       ,A.LAST_NAME;
```

Fix a bad join challenge - Text lecture

You have been given the following query to produce the selected columns but the join is not working correctly. Run the following query and then fix it:

```
SELECT A.FNAME
      ,A.dob
      ,B.Game1
      ,B.Game2
      ,B.Game3
      ,B.Game4
FROM Friends A
    ,bowlers B;
```

Solution:

Pick any join you want, but for my example, we will use INNER. Also, depending on which order you put the same fiends in, you can't rely on the bowlerID and FriendID matching so I joined on names. If none of your friends and bowlers match up, insert a few friends that do.

```
SELECT A.FNAME
      ,A.dob
      ,B.Game1
      ,B.Game2
      ,B.Game3
      ,B.Game4
FROM Friends A
     INNER JOIN bowlers B
       ON A.FNAME = B.FNAME
       AND A.LNAME = B.LNAME;
```

Subselects - Text lecture

SQL allows queries inside of other quiries which can be extermely useful. Take our example from the previous lecture. If we want to know how many movies each director directed, a subselect can easily do the job.

```
SELECT A.FIRST_NAME
      ,A.LAST_NAME
      ,IFNULL(MIN(C.Release_Year),'N/A') AS FirstDirected
      ,IFNULL(MAX(B.Release_Year),'N/A') AS LatestRelease
      ,D.cnt AS MoviesDirected
```

```
FROM people A
    LEFT OUTER JOIN movies B
     ON B.director = A.ActorID
    LEFT OUTER JOIN movies C
     ON C.director = B.director
    LEFT OUTER JOIN (SELECT Z.Director AS ActorID
                          ,COUNT(Z.MovieID) AS cnt
                    FROM movies Z
                    GROUP BY Z.Director) D
     ON D.ActorID = A.ActorID
GROUP BY A.FIRST_NAME
       ,A.LAST_NAME
       ,D.cnt
ORDER BY D.cnt DESC;
```

While there are performance hits from subselects in the SELECT section, it also is an option:

```
SELECT A.FIRST_NAME
     ,A.LAST_NAME
     ,IFNULL(MIN(C.Release_Year),'N/A') AS FirstDirected
     ,IFNULL(MAX(B.Release_Year),'N/A') AS LatestRelease
     ,(SELECT COUNT(Z.MovieID)
       FROM movies Z
       WHERE Z.Director = A.ActorID) AS MoviesDirected
FROM people A
    LEFT OUTER JOIN movies B
     ON B.director = A.ActorID
    LEFT OUTER JOIN movies C
     ON C.director = B.director
GROUP BY A.FIRST_NAME
       ,A.LAST_NAME
ORDER BY MoviesDirected DESC;
```

More subselects - Text lecture

SUBSELECTS are also useful when you want to refer to a table but don't plan on returning results from that additional table.

Let's try to pull a list of people who have directed movies, but we don't care to display anything from the movies.

```
SELECT A.FIRST_NAME
     ,A.LAST_NAME
FROM people A
WHERE EXISTS (SELECT 1
           FROM movies Z
           WHERE Z.Director = A.ActorID)
GROUP BY A.FIRST_NAME
      ,A.LAST_NAME;
```

Notice that they SELECT is set to return 1. You need to have a select statement, but there is no point in specifying anything there as it will not be used.

Another alternative to accomplish the same thing is using IN. The statement Z.Director = A.ActorID is removed and replaced by putting 1 and only one column in the SELECT statement (Z.Director) and replacing 'EXISTS' with 'A.ActorID IN'

```
SELECT A.FIRST_NAME
     ,A.LAST_NAME
FROM people A
WHERE A.ActorID IN (SELECT Z.Director
                FROM movies Z)
GROUP BY A.FIRST_NAME
      ,A.LAST_NAME;
```

We can use NOT to find anyone on the People table who has never directed a movie

```
SELECT A.FIRST_NAME
     ,A.LAST_NAME
FROM people A
WHERE NOT EXISTS (SELECT 1
               FROM movies Z
               WHERE Z.Director = A.ActorID)
GROUP BY A.FIRST_NAME
      ,A.LAST_NAME;
```

Similarly, we can use NOT IN to find the same information:

```
SELECT A.FIRST_NAME
      ,A.LAST_NAME
FROM people A
WHERE A.ActorID NOT IN (SELECT Z.Director
                        FROM movies Z)
GROUP BY A.FIRST_NAME
        ,A.LAST_NAME;
```

Final Project Part 1 - Text solution

Challenge: Update with Joins

Objective: Alter the bowlers table to not keep the individual game scores. Instead we will have a Running Average score (int) based on the results from bowlresults, ensure the names of the bowlers match in both tables (FNAME and LNAME)

You will need need some new functions for this:
AVE() to pull the average of a grouped value
ROUND() will round to an integer * use google to find and see examples of these functions
Step 1, drop the unneeded columns from bowlers table:

```
ALTER TABLE bowlers
 DROP COLUMN Game1
,DROP COLUMN Game2
,DROP COLUMN Game3
,DROP COLUMN Game4;
```

Step 2: Add new column to bowlers:

```
ALTER TABLE bowlers
ADD COLUMN AveScore int DEFAULT NULL;
```

Step 3: Plan the Update

```
SELECT A.FNAME
      ,A.LNAME
      ,AVG(B.Game_Score) AS RawAveScore
      ,ROUND(AVG(B.Game_Score)) AS AveScore
FROM  bowlers A
     ,bowlResults B
WHERE A.FNAME = B.FNAME
```

```
 AND  A.LNAME = B.LNAME
GROUP BY A.FNAME
       ,A.LNAME;
```

Step 4: Create the update statement from the select

```
UPDATE bowlers A
SET AveScore = (SELECT ROUND(AVG(Game_Score)) AS AveScore
              FROM bowlResults B
              WHERE B.FNAME = A.FNAME
               AND  B.LNAME = A.LNAME);
SELECT * FROM bowlers;
```

CONGRATULATIONS!

Final Project Part 2 - Text solution

Challenge: Delete With Joins

Objective: Delete all people from the people table that have never been a director.

Step 1: Plan it out using a select Statement

```
SELECT COUNT(*) FROM people;
SELECT COUNT(*)
FROM people A
WHERE NOT EXISTS (SELECT 1
                  FROM movies Z
                  WHERE Z.Director = A.ActorID);
```

Sometimes it’s helpful to view what you are going to be deleting in terms of row count

```
SELECT *
FROM people A
WHERE NOT EXISTS (SELECT 1
                  FROM movies Z
                  WHERE Z.Director = A.ActorID);
```

Step 2: Update the Select into a Delete and Execute

```
DELETE
FROM people
```

```
WHERE NOT EXISTS (SELECT 1
                  FROM movies Z
                  WHERE Z.Director = ActorID);
```

Step 3: Confirm your results

```
SELECT * FROM people;
```