Some of the steps in the video for this particular lesson have been updated, please see the lesson notes below for the corrected version.

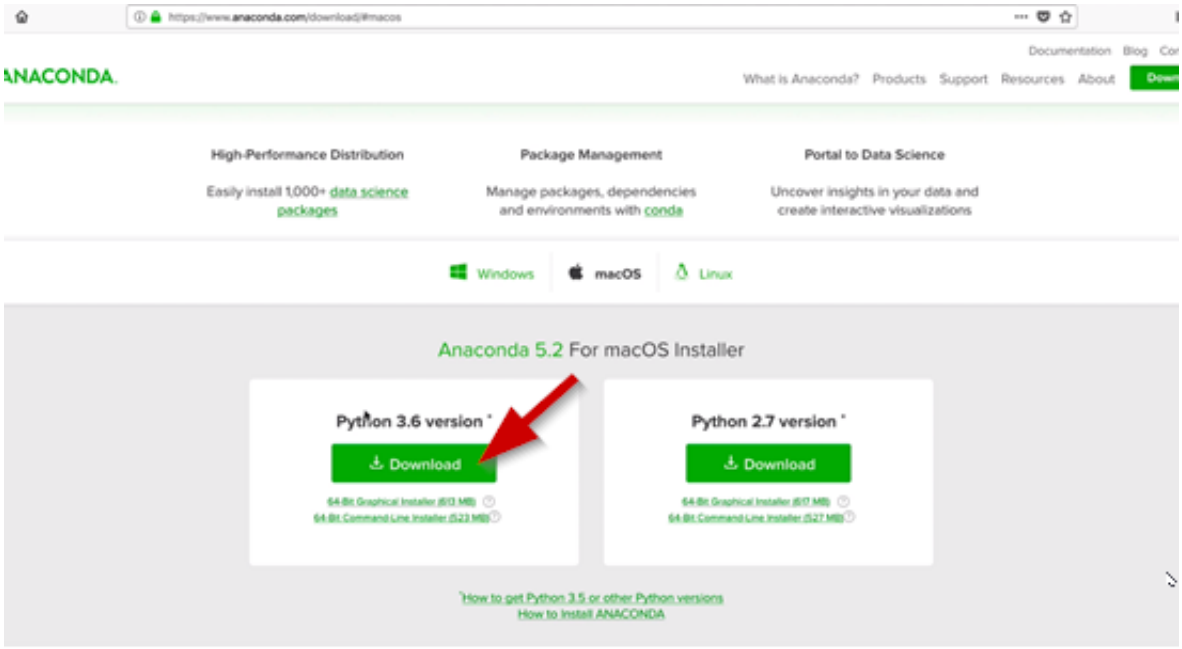
In this lesson, you will learn how to set up the development environment.

For managing our dependencies we are going to use this tool called **"Anaconda."** Essentially, you can use Anaconda to manage our dependencies and update them very easily.

You can download Anaconda from here: https://www.anaconda.com/

The direct link is here: Anaconda Download

Choose the version of Anaconda you need based on your current operating system you are using.

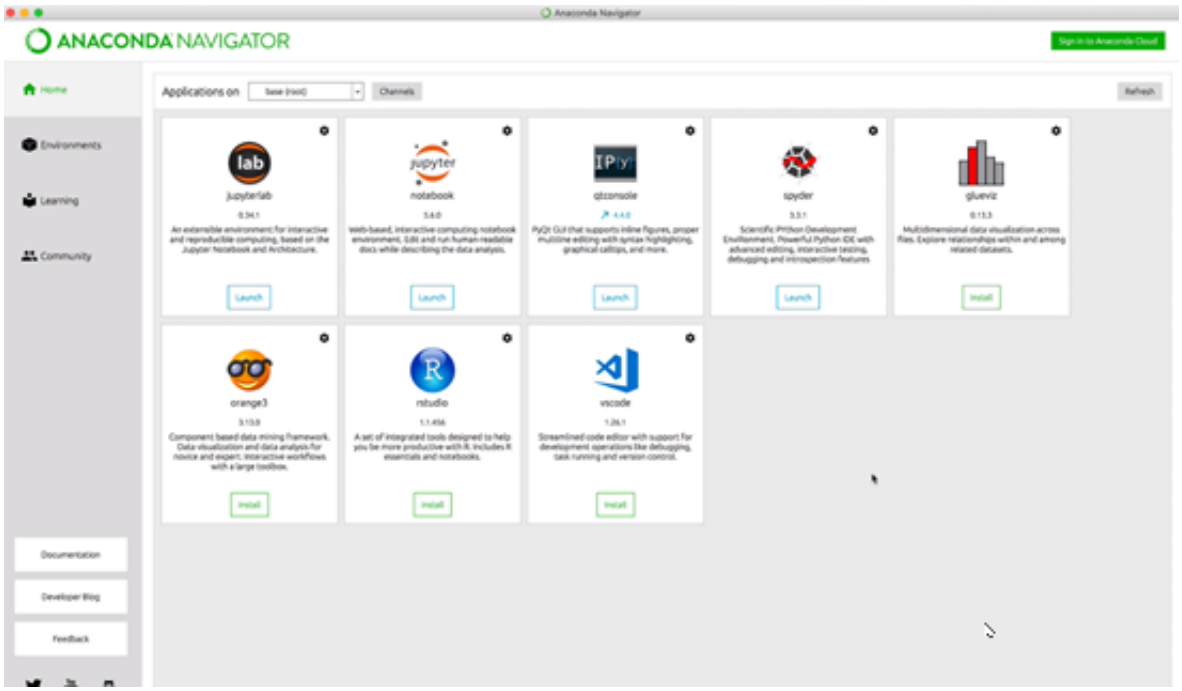


Get Started

Once you have Anaconda downloaded go ahead and follow the setup wizards instructions.

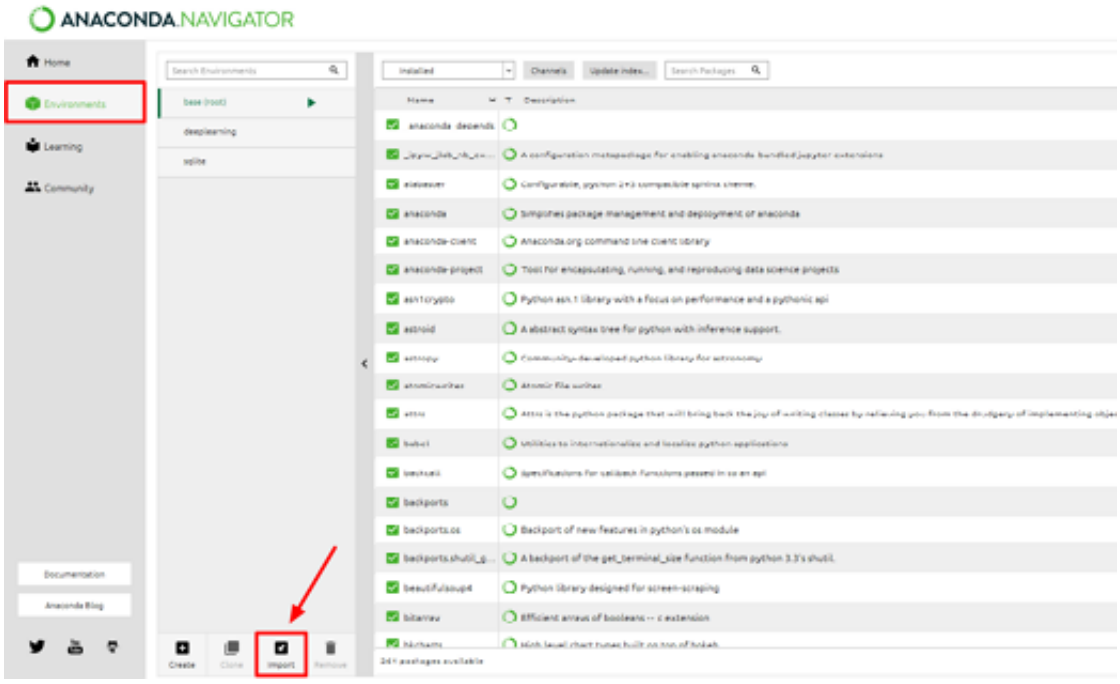
After Anaconda is installed, go ahead and open the Anaconda Navigator application:

© 2021 Zenva Pty Ltd. All rights reserved.



This already has all our dependencies bundled into these things called environments.

So **select the Environments Tab from the Anaconda menu.** At the bottom you can see a toolbar to create and **import** environments:

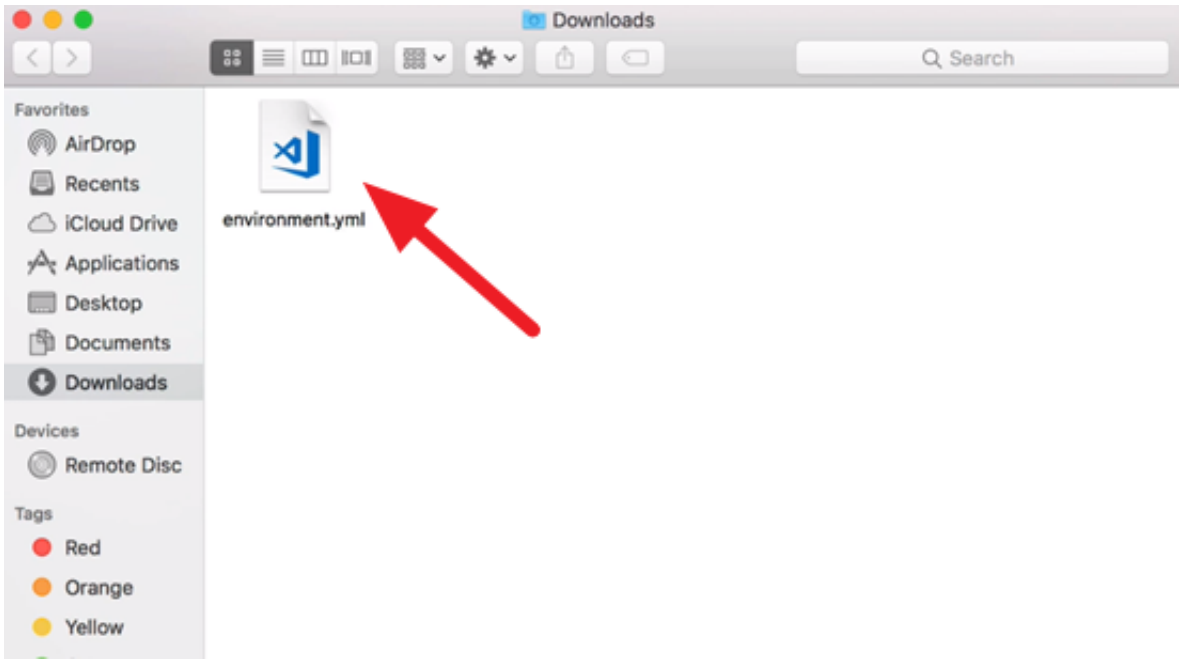


You can change your environment by clicking on one of them on the menu. After you select an environment, you can see all of the installed packages listed on the right.

The following step has been updated, and differs from the video:

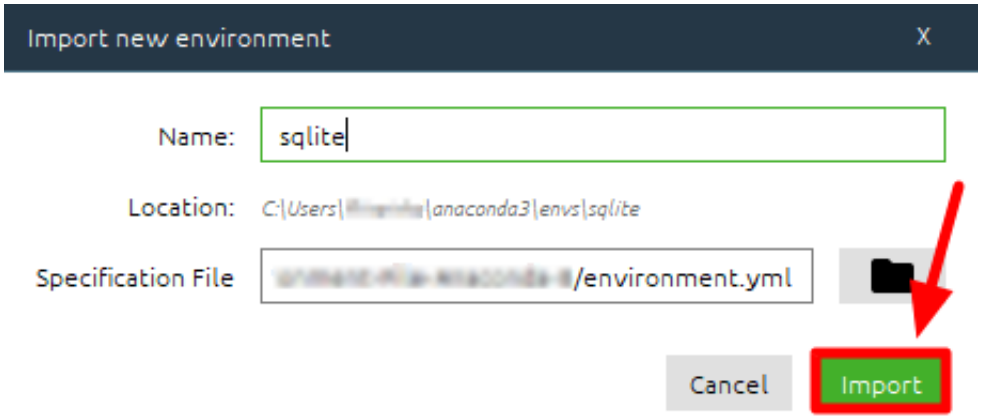
The environment we're going to use can be **downloaded** from the **Course Home page**. After you download it, you will need to **extract** the files:

© 2021 Zenva Pty Ltd. All rights reserved.



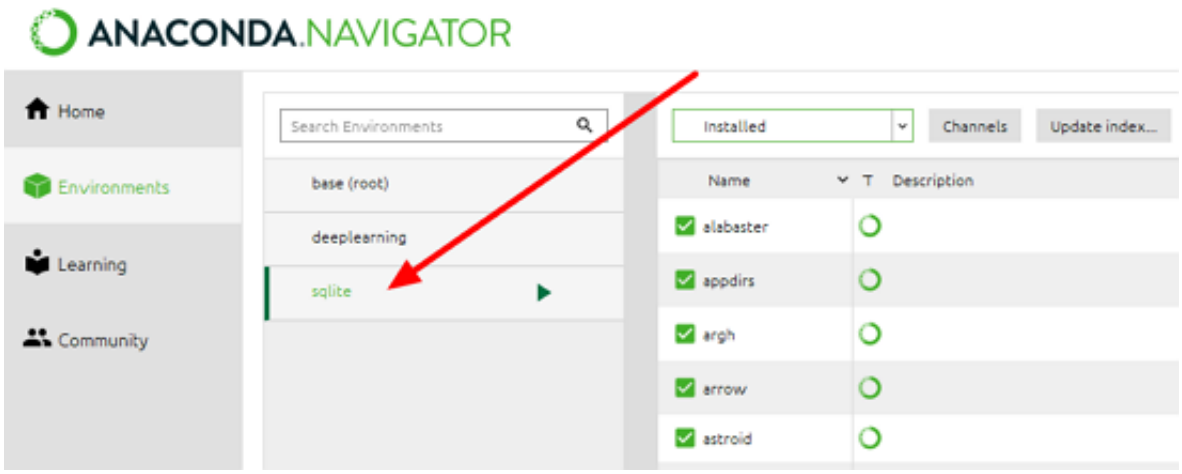
The **environment.yml** file contains all **dependencies** of the Python packages we're going to use.

All you need to do now is go to Anaconda and **import** the **environment**. It'll pull up a window for you to select the environment file you now have. **Name it "sqlite":**

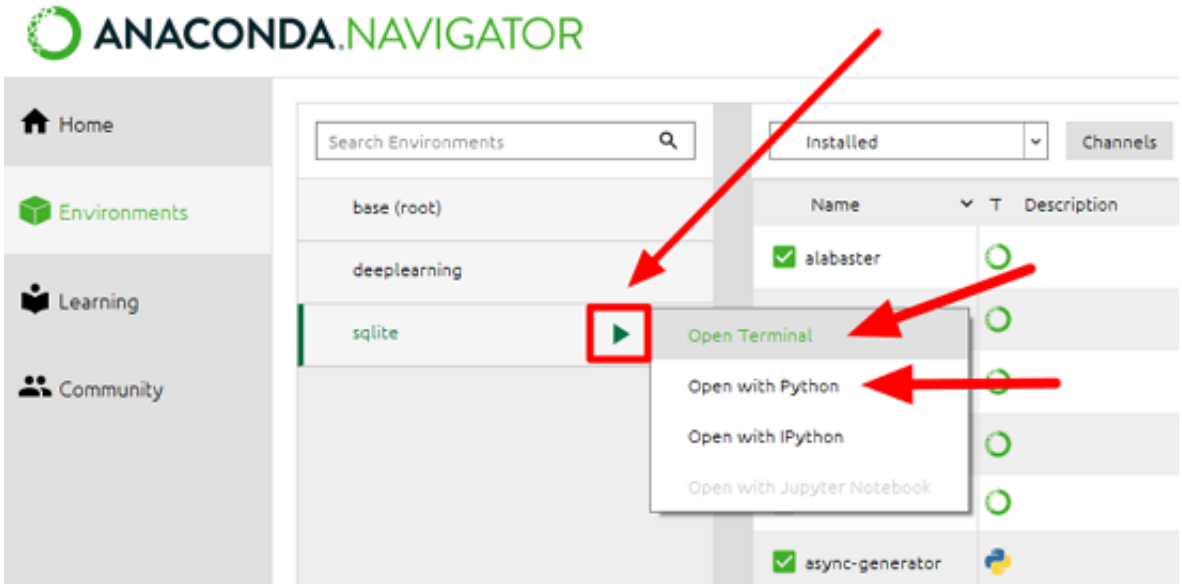


After the environment is done loading, you will now see it in the **menu** to choose:

© 2021 Zenva Pty Ltd. All rights reserved.



If you **click** on the **green arrow** next to the imported environment, you will pull up a **drop-down menu,** and from it you can **open a terminal** or a Python interpreter:



© 2021 Zenva Pty Ltd. All rights reserved.

In this lesson, we will look at how to create our own data structure in a *sqlite* database. So far we have looked at how to run SQL statements against a database that already has tables and data. Now we are going to look at how to create our own database and tables. Lets look at the syntax, various column data types.

Lets start by writing the SQL statement to create a table. We are writing this code inside a text editor. I am using Visual Studio Code (https://code.visualstudio.com/) which can be installed as part of Anaconda (https://docs.anaconda.com/anaconda/user-guide/tasks/integration/vscode/). However any good text editor should suffice.

Using our text editor, we create a file called *create-table.sql* in the same directory that contains our database (the one we have been using thus far).

```
Mohits-MacBook-Pro-2:sqlite presenter$ ls
chinook.db              create-tables.sql       track-data.csv          tracks-and-albums.sql
Mohits-MacBook-Pro-2:sqlite presenter$
```

Lets now create a script to create our own database. Since this is a student database, we might need columns such as a unique student id for every student , name, Gpa, Email.

Data Types and Constraints

Let us look at the detailed requirements of the columns in a **Student** table.

Column Name	Data Type Requirements	Sqlite Data Type	NULL allowed	UNIQUE ?	Notes
StudentId	Number	INTEGER	NO	YES	Student Id is unique to each student
FirstName	String	TEXT	NO	NO	Each student must have a first name
MiddleName	String	TEXT	YES	NO	Optional
LastName	String	TEXT	NO	NO	Each student must have a last name
Gpa	Floating point decimal number	REAL	NO	NO	Each student must have a GPA score
Email	String	TEXT	NO	YES	Each student is assigned a unique email id

NULL Values

It may not be be necessary to disallow NULL values in all columns. Lets look at the various columns. **StudentId** clearly cannot be NULL since each student must have a valid and unique id.

Likewise for the **FirstName** field. The **MiddleName** field can be NULL for some students. **LastName** should not be NULL. Each student must have a **Gpa** which cannot be NULL. **Email** can

© 2019 Zenva Pty Ltd. All rights reserved.

also not be NULL as the university will assign an email to each student.

PRIMARY KEY

StudentId is a special column as it uniquely identifies each student row in the table via a number. This is what we call a PRIMARY KEY. The convention is to have the Primary Key of the table to be set on an Id column of the table e.g. **StudentId** on the Student table.

Comments

Comments in a SQL script either begin with a '–' or are enclosed inside /* comment */.

Keeping the above constraints in mind, let us write our SQL script to create the STUDENT table:

```
-- create a table Student
CREATE TABLE IF NOT EXISTS Student (
	StudentId INTEGER PRIMARY KEY, -- not null, uniquely identifies  a row in this
 table.
	FirstName TEXT NOT NULL,    -- do not allow null values when inserting into th
is columns.
	MiddleName TEXT,
	LastName TEXT NOT NULL,
	Gpa REAL NOT NULL,     -- floating point decimal number.
	Email TEXT NOT NULL UNIQUE,       -- each email id must be unique
);
```

NOT EXIST

The IF NOT EXIST clause helps especially if we are running this script again and again. The table will not be created if it already exists.

CHALLENGE

Create a table called **Professor** with the columns **EmployeeId** (Integer, Primary Key), **FirstName**, **LastName**, **Email**. Follow the same constraints as in the case of the **Student** table. Save this SQL in the same file as above i.e. *create-tables.sql*.

Ok we're back.

```
CREATE TABLE IF NOT EXISTS Professor (
	EmployeeId INTEGER PRIMARY KEY, -- not null, uniquely identifies  a row in thi
s table.
	FirstName TEXT NOT NULL,    -- do not allow null values when inserting into th
is columns.
	LastName TEXT NOT NULL,
	Email TEXT NOT NULL UNIQUE,       -- each email id must be unique
);
```

Lets now run the above SQL statements using *sqlite*. We continue to use the same directory as before.

© 2019 Zenva Pty Ltd. All rights reserved.

```
$ sqlite university.db

sqlite>  .read create-tables.sql
sqlite> .tables
sqlite> .schema
```

```
Mohits-MacBook-Pro-2:sqlite presenter$ ls
chinook.db              create-tables.sql       track-data.csv          tracks-and-albums.sql
Mohits-MacBook-Pro-2:sqlite presenter$ sqlite3 university.db
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
sqlite> .read create-tables.sql
sqlite> .tables
Professor  Student
sqlite> .schema
CREATE TABLE Student(
    StudentId INTEGER PRIMARY KEY, -- unique, not null value used to uniquely identify a row in th
is table
    FirstName TEXT NOT NULL, -- do not allow NULL values when inserting into this column
    MiddleName TEXT,
    LastName TEXT NOT NULL,
    Gpa REAL NOT NULL, -- floating point, decimal number
    Email TEXT NOT NULL UNIQUE
);
CREATE TABLE Professor(
    EmployeeId INTEGER PRIMARY KEY,
    FirstName TEXT NOT NULL,
    LastName TEXT NOT NULL,
    Email TEXT NOT NULL UNIQUE
);
```

To verify that **university.db** actually contains our database, let us disconnect and load up the database again.

```
$ ^D
$ sqlite3 university.db

sqlite> .schema
```

© 2019 Zenva Pty Ltd. All rights reserved.

```
sqlite> ^D
Mohits-MacBook-Pro-2:sqlite presenter$ ls
chinook.db              track-data.csv          university.db
create-tables.sql       tracks-and-albums.sql
Mohits-MacBook-Pro-2:sqlite presenter$ sqlite3 university.db
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE Student(
    StudentId INTEGER PRIMARY KEY, -- unique, not null value used to uniquely identify a row in th
is table
    FirstName TEXT NOT NULL, -- do not allow NULL values when inserting into this column
    MiddleName TEXT,
    LastName TEXT NOT NULL,
    Gpa REAL NOT NULL, -- floating point, decimal number
    Email TEXT NOT NULL UNIQUE
);
CREATE TABLE Professor(
    EmployeeId INTEGER PRIMARY KEY,
    FirstName TEXT NOT NULL,
    LastName TEXT NOT NULL,
    Email TEXT NOT NULL UNIQUE
);
```

We can now see the tables we created earlier.

In summary, this is how we create tables in a database using *sqlite*.

© 2019 Zenva Pty Ltd. All rights reserved.

In this video, we are going to look at how to insert rows into a specific table. We will continue to work with the *university.db* database (in which we have created two tables). Lets look at the syntax to insert rows into a table.

```
$ sqlite3 university.db
sqlite> .tables
sqlite> .schema Student
```

```
Mohits-MacBook-Pro-2:sqlite presenter$ sqlite3 university.db
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
sqlite> .tables
Professor  Student
sqlite> .schema Student
CREATE TABLE Student(
    StudentId INTEGER PRIMARY KEY, -- unique, not null value used to uniquely identify a row in th
is table
    FirstName TEXT NOT NULL, -- do not allow NULL values when inserting into this column
    MiddleName TEXT,
    LastName TEXT NOT NULL,
    Gpa REAL NOT NULL, -- floating point, decimal number
    Email TEXT NOT NULL UNIQUE
);
```

Lets add some students to our **Student** table.

```
sqlite> INSERT INTO Student (FirstName, LastName, Gpa, Email) VALUES ('Mohit', 'Deshp
ande', 4.0, 'mdeshpande@university.edu');
```

NOTES :

1. We are going to let *sqlite* assign a numeric value to the StudentId field which is a PRIMARY KEY. So we don't assign it a value in the INSERT statement.

2. The MiddleName field is not a required field and we omit it in this INSERT statement.

```
sqlite> .nullvalue NULL
```

Lets verify that this record actually got inserted in the **Student** table.

```
sqlite> SELECT * FROM Student;
```

```
sqlite> SELECT * FROM Student;
1|Mohit|NULL|Deshpande|4.0|mdeshpande@university.edu
```

© 2019 Zenva Pty Ltd. All rights reserved.

The INSERT has been successful. Notice that *sqlite* has automatically assigned a value of **1** to the **StudentId** PRIMARY KEY column. This column will auto increment the next time we insert another record into the **Student** table.

CHALLENGE : Insert another student record into the Student table using the same INSERT syntax. Use any name you want.

Ok we're back.

```
sqlite> INSERT INTO Student (FirstName, LastName, Gpa, Email) VALUES ('John', 'Doe',
4.0, 'jdoe@university.edu');

sqlite> SELECT * FROM Student;
```

```
sqlite> INSERT INTO Student(FirstName, LastName, Gpa, Email) VALUES('John', 'Doe', 4.0, 'jdoe@univ
ersity.edu');
sqlite> SELECT * FROM Student;
1|Mohit|NULL|Deshpande|4.0|mdeshpande@university.edu
2|John|NULL|Doe|4.0|jdoe@university.edu
```

We can see another student in the table.

Lets add some data to the **Professor** table.

```
sqlite> .schema Professor
```

```
sqlite> .schema Professor
CREATE TABLE Professor(
    EmployeeId INTEGER PRIMARY KEY,
    FirstName TEXT NOT NULL,
    LastName TEXT NOT NULL,
    Email TEXT NOT NULL UNIQUE
);
```

```
sqlite> INSERT INTO Professor (FirstName, LastName, Email) VALUES ('Richard', 'Feynma
n', 'rfeynman@university.edu');

sqlite> INSERT INTO Professor (FirstName, LastName, Email) VALUES ('Albert', 'Einstei
n', 'aeinstein@university.edu');

sqlite> SELECT * FROM Professor;
```

© 2019 Zenva Pty Ltd. All rights reserved.

```
sqlite> INSERT INTO Professor(FirstName, LastName, Email) VALUES('Richard', 'Feynman', 'rfeynman@u
niversity.edu');
sqlite> INSERT INTO Professor(FirstName, LastName, Email) VALUES('Albert', 'Einstein', 'aeinstein@
university.edu');
sqlite> SELECT * FROM Professor;
1|Richard|Feynman|rfeynman@university.edu
2|Albert|Einstein|aeinstein@university.edu
```

We see two records in the **Professor** table. Both of them have unique id's assigned automatically by *sqlite*.

In summary, the INSERT statement can be used to insert rows into a table. A column which is designated as the PRIMARY KEY will automatically assigned a unique value by *sqlite*.

© 2019 Zenva Pty Ltd. All rights reserved.

In this video, lets take a look at at how to update values in a table using the UPDATE statement.

We connect to our database.

```
$ sqlite3 university.db

sqlite> .table
```

```
sqlite> .table
Professor  Student
sqlite>
```

```
sqlite> SELECT * FROM Student;
```

```
sqlite> SELECT * FROM Student;
1|Mohit|NULL|Deshpande|4.0|mdeshpande@university.edu
2|John|NULL|Doe|4.0|jdoe@university.edu
```

I want to update my GPA (corresponding to StudentId :1) to something else.

```
sqlite> UPDATE Student SET Gpa = 3.875 ;
```

```
sqlite> UPDATE Student SET Gpa=3.875;
sqlite> SELECT * FROM Student;
1|Mohit|NULL|Deshpande|3.875|mdeshpande@university.edu
2|John|NULL|Doe|3.875|jdoe@university.edu
```

Without a WHERE clause, sqlite updates ALL Gpas to 3.875. This is not the intended result. Let us modify the UPDATE statement above to set John Doe's Gpa back to 4.0.

```
sqlite> UPDATE Student SET Gpa = 4.0 WHERE StudentId = 2;

sqlite> SELECT * FROM Student;
```

© 2019 Zenva Pty Ltd. All rights reserved.

```
sqlite> UPDATE Student SET Gpa=4.0 WHERE StudentId=2;
sqlite> SELECT * FROM Student;
1|Mohit|NULL|Deshpande|3.875|mdeshpande@university.edu
2|John|NULL|Doe|4.0|jdoe@university.edu
```

We see that John Doe's Gpa has changed back to 4.0.

We can use the IN clause in the WHERE clause to update several rows at a time.

```
sqlite> UPDATE Student SET Gpa = 4.0 WHERE StudentId IN (1, 2);

sqlite> SELECT * FROM Student;
```

```
sqlite> UPDATE Student SET Gpa=3.945 WHERE StudentId IN (1, 2);
sqlite> SELECT * FROM Student;
1|Mohit|NULL|Deshpande|3.945|mdeshpande@university.edu
2|John|NULL|Doe|3.945|jdoe@university.edu
```

CHALLENGE : Write an UPDATE statement that will update just Mohit Deshpande's Gpa to a 4.0.

```
sqlite> UPDATE Student SET Gpa = 4.0 WHERE StudentId =1 ;
```

```
sqlite> UPDATE Student SET Gpa=4.0 WHERE StudentId=1;
sqlite> SELECT * FROM Student;
1|Mohit|NULL|Deshpande|4.0|mdeshpande@university.edu
2|John|NULL|Doe|3.945|jdoe@university.edu
```

Another way to achieve the above update would have been to update the record containing Email: mdeshpande@university.edu.

```
sqlite> UPDATE Student SET Gpa = 4.0 WHERE Email = ' mdeshpande@university.edu' ;
```

I could also have used the following to get the same effect.

```
sqlite> UPDATE Student SET Gpa = 4.0 WHERE FirstName = 'Mohit' ;
```

This will work in this case as I am the only student with FirstName = 'Mohit'. However if there were other rows in the table with the same FirstName, they would also get updated with the same Gpa.

In summary, this is how we update rows in a particular table using the UPDATE statement. The WHERE clause helps us narrow down the exact set of rows that we want to update. The SET clause allows us to set the value of a specific column to a specific value.

© 2019 Zenva Pty Ltd. All rights reserved.

In this video, we will look at how to delete rows from a table. The DELETE statement is similar to an UPDATE statement except for the fact that we are not changing column values in the table.

Suppose I recently graduated from university and there was no need to keep my information in the **Student** table.

We connect to our database.

```
$ sqlite3 university.db

sqlite> SELECT * FROM Student;
```

```
sqlite> SELECT * FROM Student;
1|Mohit|NULL|Deshpande|4.0|mdeshpande@university.edu
2|John|NULL|Doe|3.945|jdoe@university.edu
```

Lets get rid of my record (StudentId:1) from the **Student** table.

```
sqlite> DELETE FROM Student WHERE StudentId = 1;

sqlite> SELECT * FROM Student;
```

```
sqlite> DELETE FROM Student WHERE StudentId=1;
sqlite> SELECT * FROM Student;
2|John|NULL|Doe|3.945|jdoe@university.edu
```

We see that my record has been removed from the table. If we make the WHERE clause more general, we can delete more rows from the table.

```
sqlite> DELETE FROM Student WHERE StudentId =IN (1, 2) ;
```

The above statement will delete all students whose StudentId's are in the IN list.

Suppose we want to delete ALL rows from the Student table. We want to keep the table around while deleting all the data from it. We simply ignore the WHERE clause.

```
sqlite> DELETE FROM Student;

sqlite> SELECT * From Student;
```

© 2019 Zenva Pty Ltd. All rights reserved.

```
sqlite> DELETE FROM Student;
sqlite> SELECT * FROM Student;
sqlite>
```

Notice we don’t get any results back as all rows have been deleted from the table. This is somewhat similar to the UPDATE statement (without the WHERE clause) in the last video where all rows got updated with the value specified in the SET clause. The table still exists in the database after we run this statement.

In summary, this is how rows get deleted from a table. We use the WHERE clause when we want to restrict which rows get deleted and remove the WHERE clause completely when we want ALL rows deleted.

© 2019 Zenva Pty Ltd. All rights reserved.

In this video, lets look at how to delete tables from a database.

In the last video, we looked at how to delete a table using the DELETE FROM <table name> syntax. This statement would delete all rows from the table but keep the table intact.

```
sqlite> SELECT * FROM Student;

sqlite> .table

sqlite> .schema Student
```

```
sqlite> SELECT * FROM Student;
sqlite> .table
Professor  Student
sqlite> .schema Student
CREATE TABLE Student(
    StudentId INTEGER PRIMARY KEY, -- unique, not null value used to uniquely identify a row in th
is table
    FirstName TEXT NOT NULL, -- do not allow NULL values when inserting into this column
    MiddleName TEXT,
    LastName TEXT NOT NULL,
    Gpa REAL NOT NULL, -- floating point, decimal number
    Email TEXT NOT NULL UNIQUE
);
```

The **Student** table still exists.

Lets get rid of the table itself.

```
sqlite> DROP Table Student;
```

```
sqlite> DROP TABLE Student;
sqlite> .table
Professor
```

We only see the **Professor** table.

```
sqlite> .schema Student
```

There is no output as there is no table by name **Student** anymore.

NOTE : If you have any data inside a table and then issue the DROP TABLE command on it, you will LOSE all the data.

© 2019 Zenva Pty Ltd. All rights reserved.

In summary, we can use the DROP TABLE command to delete tables from the database. All data in the table will be deleted along with the table itself.

© 2019 Zenva Pty Ltd. All rights reserved.

In this video, let us look the ALTER TABLE command to modify tables. We can use this command to rename tables and to add columns to a table.

```
sqlite> .table
Professor  Studentt
sqlite> .schema Studentt
CREATE TABLE Studentt(StudentId INTEGER PRIMARY KEY, FirstName TEXT NOT NULL, LastName TEXT NOT NU
LL, Gpa REAL NOT NULL, Email TEXT NOT NULL UNIQUE);
```

I have recreated my student table but misspelt it as **Studentt**. I also forgot to add the **MiddleName** column to the table. Lets correct these using the ALTER TABLE command.

```
sqlite> ALTER TABLE Studentt RENAME TO Student;

sqlite> .table
```

```
sqlite> ALTER TABLE Studentt RENAME TO Student;
sqlite> .table
Professor  Student
```

We can see that the table has been renamed to **Student**.

Lets look at the data in the table.

```
sqlite> SELECT * FROM Student;
```

```
sqlite> SELECT * FROM Student;
1|Mohit|Deshpande|4.0|mdeshpande@university.edu
```

Lets now add the **MiddleName** column to the table.

```
sqlite> ALTER TABLE Student ADD COLUMN MiddleName TEXT;

sqlite> .schema Student
```

```
sqlite> ALTER TABLE Student ADD COLUMN MiddleName TEXT;
sqlite> .schema Student
CREATE TABLE IF NOT EXISTS "Student"(StudentId INTEGER PRIMARY KEY, FirstName TEXT NOT NULL, LastN
ame TEXT NOT NULL, Gpa REAL NOT NULL, Email TEXT NOT NULL UNIQUE, MiddleName TEXT);
sqlite> SELECT * FROM Student;
1|Mohit|Deshpande|4.0|mdeshpande@university.edu|NULL
```

We see that the MiddleName column has been added. We also see that the existing data rows have NULL for the MiddleName column. If we want to add a column with a NOT NULL constraint, we would

© 2019 Zenva Pty Ltd. All rights reserved.

have to specify default values which would get added to that column for all rows in the table. We are not going to get into that right now.

In summary, there are two kinds of modifications we can make to an existing table. We can rename it and add columns to it.

© 2019 Zenva Pty Ltd. All rights reserved.

In this video, we will use our existing knowledge of various commands to remove a column from a table. We will also learn a new feature supported by *sqlite* called transactions.

Suppose I want to remove the MiddleName column from the **Student** table.

Transactions

Let look at transactions by writing one. Lets open our favorite code editor and create a new file *remove-column.sql*.

```
1  -- remove MiddleName from Student table
2
3  -- inside of the transaction, either ALL of the SQL runs successfully,
4  -- or our database rolls back to before the transaction
5  BEGIN TRANSACTION;
6
7  -- 1. Rename our table to a temp name
8  -- 2. Create a new table with the column removed
9  -- 3. Copy data from the old, temp table into the new table
10 -- 4. Get rid of the old, temp table
11
12 COMMIT;
```

A transaction is a set of statements inside of a BEGIN TRANSACTION – COMMIT block. This causes either ALL of the statements inside the block to run successfully and commit or causes ALL of them to fail and rollback the transaction. This is to say that if any error occurred inside the block, the state of the database would be rolled back to what existed before the transaction began.

This behavior prevents the database from being in an inconsistent state with half the statements having succeeded and half having failed.

```
BEGIN TRANSACTION;

-- 1. Rename our table to a temp name

ALTER TABLE Student RENAME To Student_Temp;

-- 2. Create a new table with the MiddleName column removed

CREATE TABLE Student(

  StudentId INTEGER PRIMARY KEY,

  FirstName TEXT NOT NULL,

  LastName TEXT NOT NULL,

  Gpa REAL NOT NULL,

  Email TEXT NOT NULL

);
```

© 2019 Zenva Pty Ltd. All rights reserved.

```
--3. Copy data from the old, temp table into the new table using INSERT INTO – SELECT

-- The results of the SELECT statement are inserted into Student table.

INSERT INTO Student

SELECT StudentId, FirstName, LastName, Gpa, Email

FROM Student_Temp;

--4. Get rid of the old, temp table

DROP TABLE Student_Temp;

COMMIT;
```

It should be clearer why we are enclosing these statements into a TRANSACTION. Suppose the CREATE TABLE statement above failed with a error, then the database would be left in an inconsistent state. If there were many statements, it would have been difficult to track down which statement actually failed. This makes the TRANSACTION block really useful.

Lets now run this SQL script.

```
sqlite> .schema Student

sqlite> SELECT * FROM Student;

sqlite> .read remove-column.sql

sqlite> .schema Student

sqlite> SELECT * FROM Student;
```

© 2019 Zenva Pty Ltd. All rights reserved.

```
Mohits-MacBook-Pro-2:sqlite presenter$ ls
chinook.db              remove-column.sql       tracks-and-albums.sql
create-tables.sql       track-data.csv          university.db
Mohits-MacBook-Pro-2:sqlite presenter$ sqlite3 university.db
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
sqlite> .read remove-column.sql
sqlite> .schema Student
CREATE TABLE Student(
    StudentId INTEGER PRIMARY KEY,
    FirstName TEXT NOT NULL,
    LastName TEXT NOT NULL,
    Gpa REAL NOT NULL,
    Email TEXT NOT NULL UNIQUE
);
sqlite> SELECT * FROM Student;
1|Mohit|Deshpande|4.0|mdeshpande@university.edu
```

Notice we don't have the MiddleName column any more as the CREATE TABLE statement does not have it in its column list. The SELECT statement that picks data from the **Student_Temp** table also leaves this column out.

The data from the previous table has also been successfully copied over to the new Student table.

In summary, we use a combination of ALTER TABLE – RENAME, CREATE TABLE, INSERT INTO – SELECT FROM to remove a column from a table. We saw how to run all the above statements inside a TRANSACTION to ensure all the above statements run successfully or fail together. We also saw how the INSERT INTO – SELECT FROM statement can be used to first select specific columns from rows from a table and insert them into another table.

© 2019 Zenva Pty Ltd. All rights reserved.

In this video, we will start to look the FOREIGN KEY constraint. Let us revisit our discussion on JOINS.

TrackId	Name	Composer	AlbumId
84	Take Five	The Dave Brubeck Quartet	1
85	Three to Get Ready	The Dave Brubeck Quartet	1
86	Strange Meadow Lark	The Dave Brubeck Quartet	1
103	So What	Miles Davis	2
104	Freddie Freeloader	Miles Davis	2
133	Original Faubus Fables	Charles Mingus	5

AlbumId	Title
1	Time Out
2	Kind of Blue
3	Breakfast Dance and Barbecue

TrackId	Name	Composer	AlbumId	Title
84	Take Five	The Dave Brubeck Quartet	1	Time Out
85	Three to Get Ready	The Dave Brubeck Quartet	1	Time Out
86	Strange Meadow Lark	The Dave Brubeck Quartet	1	Time Out
103	So What	Miles Davis	2	Kind of Blue
104	Freddie Freeloader	Miles Davis	2	Kind of Blue

Recall our discussions on the INNER JOIN. We link the **AlbumId** column of the **Track** table to the **AlbumId** of the **Album** table and link the corresponding rows together to produce the result set in the diagram above.

The values in the AlbumId field in the **Track** table correspond to an value already present in the AlbumId field in the **Album** table. What this means is that we are *not allowed* to insert a row in the Track table unless it's AlbumId value *already exists* in the Album table. In short a track has to correspond to an existing Album. This is what we call a FOREIGN KEY constraint.

Notice that the Track table has a AlbumId :5 that does not exist in the Album table. This is possible when the FOREIGN KEY constraint has not been enforced. If the constraint were enforced and we then tried to insert this row, we would get an error due to the constraint violation.

There can be different relationships between tables.

MANY: 1 Relationship

The relationship between **Track** and **Album** would be a MANY : 1 relationship i.e. Many tracks belong to 1 album.

1:MANY Relationship

An example of 1:MANY relationship is a **Course** being taught by an **Instructor** (The same instructor may teach many courses).

MANY: MANY Relationship

An example would be that a **Student** can attend multiple courses and a **Course** can be attended by multiple students.

Lets now implement a 1:MANY relationship using FOREIGN KEYS.

```
$ sqlite3 university.db

sqlite> SELECT * FROM Student;
```

© 2019 Zenva Pty Ltd. All rights reserved.

```
1|Monit|Deshpande|4.0|mdeshpande@university.edu
```

```
sqlite> SELECT * FROM Professor;
```

```
sqlite> SELECT * FROM Professor;
1|Richard|Feynman|rfeynman@university.edu
2|Albert|Einstein|aeinstein@university.edu
```

```
sqlite> .schema Professor
```

```
sqlite> .schema Professor
CREATE TABLE Professor(
    EmployeeId INTEGER PRIMARY KEY,
    FirstName TEXT NOT NULL,
    LastName TEXT NOT NULL,
    Email TEXT NOT NULL UNIQUE
);
```

Lets create a new table that holds all the courses. Lets create a new file *many-to-one.sql*.

```
-- turn on support for FOREIGN KEY in SQLite

PRAGMA foreign_key = ON;

-- many-to-
one: a course is taught by one professor but a professor may teach many courses.

CREATE TABLE Course(

  CourseId INTEGER PRIMARY KEY,

  Department TEXT NOT NULL,

  CourseNumber INTEGER NOT NULL,

  CourseName TEXT NOT NULL,

  EmployeeId INTEGER,

  FOREIGN KEY(EmployeeId) REFERENCES Professor(EmployeeId) -- FOREIGN KEY constraint
```

© 2019 Zenva Pty Ltd. All rights reserved.

```
);
```

The above FOREIGN KEY constraint ensures that every row inserted into **Course** will have a valid **EmployeeId** in the **Professor** table. There are other advanced ways to say what will happen to a professor if the course is deleted or what happens when a professor leaves the university, but we are not going to get into those scenarios right now.

Lets insert values into the Course table.

```
sqlite> .read many-to-one.sql

sqlite> .table
```

```
sqlite> .read many-to-one.sql
sqlite> .table
Course     Professor_ Student
```

Lets insert a course taught by Professor Feynman.

```
sqlite> INSERT INTO Course ( Department, CourseNumber, CourseName, EmployeeId) VALUES
(“PHYSICS”, 550, “Quantum Mechanics 1”, 1);

sqlite>
```

We don’t get any errors which means the INSERT succeeded. Lets insert another course, also taught by the same professor.

```
sqlite> INSERT INTO Course ( Department, CourseNumber, CourseName, EmployeeId) VALUES
(“PHYSICS”, 551, “Quantum Mechanics 2”, 1);

sqlite>
```

The INSERT succeeds.

Lets insert another course, this time, with an invalid EmployeeId : 5.

```
sqlite> INSERT INTO Course ( Department, CourseNumber, CourseName, EmployeeId) VALUES
```

© 2019 Zenva Pty Ltd. All rights reserved.

```
(“PHYSICS”, 551, “Quantum Mechanics 3”, 5);
```

```
sqlite> INSERT INTO Course(Department, CourseNumber, CourseName, EmployeeId) VALUES("PHYSICS", 552
, "Quantum Mechanics 3", 5);
Error: FOREIGN KEY constraint failed
```

We get an expected error since there is no professor who has EmployeeId:5 which violates the FOREIGN KEY constraint.

Lets insert a couple more courses taught by Prof. Einstein.

```
sqlite> INSERT INTO Course ( Department, CourseNumber, CourseName, EmployeeId) VALUES
(“PHYSICS”, 560, “General Relativity 1”, 2);

sqlite> INSERT INTO Course ( Department, CourseNumber, CourseName, EmployeeId) VALUES
(“PHYSICS”, 561, “General Relativity 2”, 2);

sqlite> SELECT * FROM Course;
```

```
sqlite> SELECT * FROM Course;
1|PHYSICS|550|Quantum Mechanics 1|1
2|PHYSICS|551|Quantum Mechanics 2|1
3|PHYSICS|560|General Relativity 1|2
4|PHYSICS|561|General Relativity 2|2
```

Let us now do an INNER JOIN between the **Course** and **Professor** tables and print out the names of the professors. Think about how we would do that for a moment and we will be right back.

Ok.

```
sqlite> SELECT * FROM Course INNER JOIN Professor ON Course.EmployeeId = Professor.Em
ployeeId;
```

```
sqlite> SELECT * FROM Course INNER JOIN Professor ON Course.EmployeeId = Professor.EmployeeId;
1|PHYSICS|550|Quantum Mechanics 1|1|1|Richard|Feynman|rfeynman@university.edu
2|PHYSICS|551|Quantum Mechanics 2|1|1|Richard|Feynman|rfeynman@university.edu
3|PHYSICS|560|General Relativity 1|2|2|Albert|Einstein|aeinstein@university.edu
4|PHYSICS|561|General Relativity 2|2|2|Albert|Einstein|aeinstein@university.edu
```

The query has successfully linked the EmployeeId columns in both tables and as a result we get both the course information as well as information about the professors that are teaching the courses.

Lets extract specific columns from the above JOIN.

© 2019 Zenva Pty Ltd. All rights reserved.

```
sqlite> SELECT CourseName, FirstName, LastName FROM Course INNER JOIN Professor ON Co
urse.EmployeeId = Professor.EmployeeId;
```

```
sqlite> SELECT CourseName, FirstName, LastName FROM Course INNER JOIN Professor ON Course.Employee
Id = Professo
Quantum Mechanics 1|Richard|Feynman
Quantum Mechanics 2|Richard|Feynman
General Relativity 1|Albert|Einstein
General Relativity 2|Albert|Einstein
```

The above result clearly tells us which course is taught by which professor e.g. Quantum Mechanics I is taught by Richard Feynman

In summary, given two tables in a 1:Many or Many:1 relationship, we can use FOREIGN KEY constraints to enforce that a row is inserted into a table with a foreign key only if the foreign key exists in the other table. We looked at how to create a table with a FOREIGN KEY constraint and looked at how to create a JOIN between two tables in a 1:MANY FOREIGN KEY relationship.

© 2019 Zenva Pty Ltd. All rights reserved.

In this video, we will demonstrate how to use foreign keys to create a MANY – MANY relationship.

The **Student** and **Course** tables have a MANY – MANY relationship (a student can take many courses and a course may have many students enrolled). Lets create a table to link these tables together.

Lets create a new file – *many-to-many.sql,* in our favorite editor.

```
-- turn on support for foreign keys in sqlite

PRAGMA foreign_keys = ON;

-- many-to-
many: students can attend many courses, courses can be attended by many students

CREATE TABLE Attends(

  AttendsId INTEGER PRIMARY KEY,

  StudentId INTEGER, -- links to the Student table

  CourseId INTEGER, -- links to the Course table

  FOREIGN KEY (StudentId) REFERENCES Student(StudentId),

-- CHALLENGE

-- add code to make the CourseId a FOREIGN KEY referencing the CourseId in the

-- Course table

);
```

We are back

```
CREATE TABLE Attends(

  AttendsId INTEGER PRIMARY KEY,

  StudentId INTEGER, -- links to the Student table

  CourseId INTEGER, -- links to the Course table

  FOREIGN KEY (StudentId) REFERENCES Student(StudentId),

  FOREIGN KEY (CourseId) REFERENCES Course(CourseId),

);
```

© 2019 Zenva Pty Ltd. All rights reserved.

Lets execute this script.

```
sqlite> .read many-to-many.sql

sqlite> .table

sqlite> SELECT * FROM Student;

sqlite> SELECT * FROM Course;
```

```
sqlite> .read many-to-many.sql
sqlite> .table
Attends    Course     Professor  Student
sqlite> SELECT * FROM Student;
1|Mohit|Deshpande|4.0|mdeshpande@university.edu
sqlite> SELECT * FROM Course;
1|PHYSICS|550|Quantum Mechanics 1|1
2|PHYSICS|551|Quantum Mechanics 2|1
3|PHYSICS|560|General Relativity 1|2
4|PHYSICS|561|General Relativity 2|2
```

Lets insert some rows into the **Attends** table for the above courses, student.

```
sqlite> INSERT INTO Attends(StudentId, CourseId) VALUES (1,2);

sqlite> INSERT INTO Attends(StudentId, CourseId) VALUES (1,3);
```

© 2019 Zenva Pty Ltd. All rights reserved.

```
sqlite> INSERT INTO Attends(StudentId, CourseId) VALUES(1, 2);
sqlite> INSERT INTO Attends(StudentId, CourseId) VALUES(1, 3);
sqlite> SELECT * FROM Attends;
1|1|2
2|1|3
```

Lets do some JOINS on the above tables.

```
sqlite> .schema
```

```
sqlite> .schema
CREATE TABLE Professor(
    EmployeeId INTEGER PRIMARY KEY,
    FirstName TEXT NOT NULL,
    LastName TEXT NOT NULL,
    Email TEXT NOT NULL UNIQUE
);
CREATE TABLE Student(
    StudentId INTEGER PRIMARY KEY,
    FirstName TEXT NOT NULL,
    LastName TEXT NOT NULL,
    Gpa REAL NOT NULL,
    Email TEXT NOT NULL UNIQUE
);
CREATE TABLE Course(
    CourseId INTEGER PRIMARY KEY,
    Department TEXT NOT NULL,
    CourseNumber INTEGER NOT NULL,
    CourseName TEXT NOT NULL,
    EmployeeId INTEGER,
    FOREIGN KEY (EmployeeId) REFERENCES Professor(EmployeeId) -- create the foreign key constraint
);
CREATE TABLE Attends(
    AttendsId INTEGER PRIMARY KEY,
    StudentId INTEGER,
    CourseId INTEGER,
    FOREIGN KEY (StudentId) REFERENCES Student(StudentId),
    FOREIGN KEY (CourseId) REFERENCES Course(CourseId)
);
```

Lets do a triple join, linking the **Student**, **Course**, **Attends** tables. The first INNER JOIN links **Student** and **Attends** and the second INNER JOIN links **Course** and **Attends** together.

```
sqlite> SELECT FirstName, CourseName FROM Student INNER JOIN Attends ON Student.Stude
ntId = Attends.StudentId INNER JOIN ON Course.CourseId = Attends.CourseId;
```

© 2019 Zenva Pty Ltd. All rights reserved.

```
sqlite> SELECT FirstName, CourseName FROM Student INNER JOIN Attends ON Student.StudentId = Attend
s.StudentId
   ...> INNER JOIN Course ON Course.CourseId = Attends.CourseId;
Mohit|Quantum Mechanics 2
Mohit|General Relativity 1
```

The output shows the student Mohit is taking Quantum Mechanics 2 and General Relativity I. Thus we have merged the information from two tables (**Student, Course**) in a Many – Many relationship, by using the intermediate table **Attends**. I could add another WHERE clause if I wished to filter data further.

Thus I can powerfully combine FOREIGN KEYS and JOINS to get data from several tables at the same time.

In summary, this is how we create MANY – MANY relationships between two tables in *sqlite* by creating an intermediate table which has FOREIGN KEY relationships to the two tables. In this case the **Attends** table has foreign key relationships with **Course** and **Student**.

© 2019 Zenva Pty Ltd. All rights reserved.