1. **Basics of SQL:**

   A database file (.db) stores a collection of data, known as tables. One database may contain one or many tables. Tables are a collection of data that follow a specific format. When a table is first defined, the definition includes the number of columns, the name of each column, and the type of the values in that column.

   SQL (Structured Query Language) can be used to interact with databases. There are many things SQL can do, but the most important one is the ability to make queries of databases in order to fetch data. A query is some kind of question we ask about data. For instance, if I have a table of census data, I could ask how many people live in Nebraska. If I have a table mapping everyone to their zipcodes, I could ask for a list of people who live in zipcode 15112. If I have a table of all FCE ratings, I could ask what is the average FCE rating for all courses in each department.

   This session uses sqlite3 to demonstrate SQL queries. If you do not have sqlite3 installed on your computer, here is some information about that: https://www.sqlitetutorial.net/download-install-sqlite/

   The Python file `database_maker.py`, supplied along with this PDF, should be run in a folder in order to create the file `demotables.db`. Once you have this file, you can either open a sqlite3 shell and type in SQL commands directly, or create a .sql file and run it like you would run a Python file.

   To open the shell yourself, type `sqlite3` into the terminal (make sure you are in the same directory as the database). Once there, type in the following 3 commands in order to set up the sqlite3 shell so that you can type in queries:

```
sqlite3
.open demotables.db
.header on
.mode column
```

   Once you have done that, you can type in the queries (which we will learn soon) into the shell and the results will be displayed, similar to how you can type Python code into the python interpreter and the results get printed out.

   Along with this PDF, another file called `demoqueries.sql` is provided. This file contains the same setup commands as above, as well as most of the queries used during the seminar as demonstrations, though most are commented out (two dashes form a comment in SQL). To run the file, use the following command:

```
sqlite3 < demoqueries.sql
```

   Upon doing so, any uncommented queries in the file will be executed and printed out in your terminal.

2. **Simple Queries:**

   The simplest queries are of the form `SELECT [something] FROM [table]`. The word that follows the `FROM` keyword is the table that the query will be applied to. Whatever is between the `SELECT` and `FROM` is the query itself.

   This seminar makes use of several tables stored within `demotables.db`. The first of these is the `cities` table, which is in the following format:

Name	State	Population	Elevation	Founding
Houston	TX	2304580	80	1837
San Francisco	CA	873965	52	1850
Los Angeles	CA	3898747	305	1850
...	...	...	...	...

   Below are a few simple examples of queries operating on the `cities` table:

```sql
-- Selects the entire table
SELECT *
FROM cities;

-- Selects only the first two columns
SELECT Name, State
FROM cities;
```

3. **Sorting:**

   The order of the rows can also be changed by a query which sorts them according to some metric. This is done by appending `ORDER BY [criteria]` after the `FROM` keyword. The simplest case is using a single column as a criteria, in which case the table will be sorted by that column in ascending order.

   To switch to descending order, the `DESC` keyword can be used. Also, multiple columns can be used in order to have tiebreakers (i.e. sort by column A, but if two rows have the same value for that column, then sort by column B).

   Below are a few examples of sorting queries on the `cities` table:

```sql
-- Order alphabetically by city name
SELECT *
FROM cities
ORDER BY Name ASC;

-- Order by population (highest to lowest)
SELECT *
FROM cities
ORDER BY Population DESC;

-- Order alphabetically by state, tiebreak alphabetically by city name
SELECT *
FROM cities
ORDER BY State ASC, Name ASC;

-- Order by length of cityname
SELECT *
FROM cities
ORDER BY LENGTH(Name);
```

4. **Top X Rows:**

   Another type of query is to get the top X rows according to some metric (i.e. the 5 cities with the highest population, or the 8 with the lowest elevation). This can be done by first sorting with `ORDER BY`, then appending `LIMIT X` to the end of the query. This will chop off all but the first X rows.

   NOTE: there are many different versions of SQL, and their syntaxes have some difference. This particular feature is one of the ones that varies the most from version-to-verison. The other versions do the same thing, but the syntax for it often looks quite different.

   Below are a few examples of Top X queries on the `cities` table:

```sql
-- Get the 5 cities with the highest altitude
SELECT *
FROM cities
ORDER BY Elevation DESC
LIMIT 5;

-- Get the 10 cities that were founded first
SELECT *
FROM cities
ORDER BY Founding ASC
LIMIT 10;
```

5. **Unique Rows:**

   There are many types of statements in SQL besides just `SELECT`. We won't go into most of them, and are focusing on making queries. But there is another type of query keyword: `SELECT DISTINCT`. These types of queries work the same as `SELECT` queries, except they remove duplicates from the final output.

   Below are a few examples of finding unique entries from the `cities` table:

```sql
-- Get a list of unique states
SELECT DISTINCT State
FROM cities
ORDER BY State ASC;

-- Get a list of unique starting letters of city names
SELECT DISTINCT SUBSTR(Name, 1, 1)
FROM cities
ORDER BY Name;

-- Get a list of unique city name lengths
SELECT DISTINCT LENGTH(Name)
FROM cities
ORDER BY LENGTH(Name);
```

6. **Filtering Rows:**

   Queries can also filter the rows of the table in order to obtain a subset of the rows that obey a certain criteria. This is done with the `WHERE`, followed by one or more conditions. These conditions can include comparisons (equal, less than, etc.) as well as boolean operators (and, or, etc.).

   There are many possible conditions, but below are a few examples of filtering from the `cities` table:

```
-- Get the cities where the elevation is at most 50 feet
SELECT *
FROM cities
WHERE Elevation <= 50;

-- Get the cities founded in the 1700s
SELECT *
FROM cities
WHERE Founding BETWEEN 1700 AND 1799

-- Get the cities in California
SELECT *
FROM cities
WHERE State = "CA";

-- Get the cities with population above 1000000 and elevation below 500
SELECT *
FROM cities
WHERE Population > 1000000 AND Elevation < 500;
```

7. **Filtering (Advanced):**

   A slightly more advanced trick with filtering is the use of the `LIKE` keyword, which allows values in a certain column to be compared against a pattern to see if they "match". These patterns are strings where the characters need to exactly match the column value for the string to match, but with two special cases. An underscore character can match with any single character, and a percent sign can match with any number of characters of any type. For example, the pattern "Q%A_" is a pattern that matches with any string where the first character is a Q and the 2nd to last character is an A (like the word Quizzical).

   Below are a few examples of using this behavior on the `cities` table:

```
-- Get the cities from a state that starts with N
SELECT *
FROM cities
WHERE State LIKE "N_";

-- Get the cities where the name starts with San
SELECT *
FROM cities
WHERE State LIKE "San%";

-- Get the cities where the name contains at least two O's
SELECT *
FROM cities
WHERE Name LIKE "%o%o%";

-- Get the cities that were founded at the start of a decade
SELECT *
FROM cities
WHERE Founding LIKE "%0";
```

8. **Operations:**

   Besides just retrieving data, queries can also transform the data by applying various transformations. These include performing mathematical operations on the data, calling a few builtin functions on it, renaming the columns using the `AS` keyword, or even aggregate operations that combine the rows in various ways.

   Below are a few examples of using operations on the `cities` table:

```sql
-- Extract the century of each city's founding
SELECT Name, 100*(Founding/100)
FROM cities;

-- Same as above, with the columns renamed
SELECT Name AS City_Name,
       100*(Founding/100) AS Founding_Century
FROM cities
ORDER BY Founding_Century;

-- Get the sum of the populations, and the average elevation
SELECT SUM(Population) AS Total_Population,
       ROUND(AVG(Elevation)) AS Average_Elevation
FROM cities;
```

9. **Grouping Rows:**

   Those aggregate operations mentioned earlier become even more useful when we cluster rows together into various groupings. The `GROUP BY` keyword groups the rows together based on a certain criteria (similar to how `ORDER BY` sorts them by some criteria), at which point aggregate operations can be used on each of the groupings. This can be thought of as "squishing" all the rows with the same values for that criteria into a single super-row, about which we can extract information.

   Below are a few examples of grouping together the rows of the `cities` table:

```sql
-- Group the cities by state
SELECT *
FROM cities
GROUP BY State;

-- Count the number of cities per state
SELECT State, Count(*) as Num_Cities
FROM cities
GROUP BY State;

-- Count the number of cities and population by state starting letter
SELECT SUBSTR(State, 1, 1) AS Letter,
       COUNT(*) AS Count,
       SUM(Population) AS Pop
FROM cities
GROUP BY Letter;

--  Pick the largest city from within each state
SELECT Name, State, MAX(Population) As Population
FROM cities
GROUP BY State;
```

Before introducing the final topic, it is time to introduce another table used in this seminar: the `phone_book` table. This table contains rows describing fictional people, and is in the following format:

First	Last	City	Zipcode	Phone
Richard	Rodriguez	Los Angeles	39256	350-680-6155
Catherine	Lee	Tampa	38221	259-758-8517
Raymond	Rodriguez	Chicago	23434	854-862-9830
...	...	...	...	...

10. **Joining Tables:**

The last major subject we will cover is use of `INNER JOIN`, which enables more complex queries that combine data from multiple tables. There are several types of join, but we'll stick with the simplest version.

The basic idea is that instead of just selecting from a a single table, we select from the join of two tables and can then use our usual queries (sorting, grouping, filtering, etc.) on this combined table. The syntax for this is `SELECT [something] FROM [table 1] INNER JOIN [table 2] ON [criteria].`

Since there are two tables, and they could have column names that clash, column names are instead referenced by prefacing them with the table name (i.e. `cities.Name` instead of `Name`).

The way that the two tables are joined will depend on the criteria used to join them. `INNER JOIN` will look for any pair of rows from the two tables that meet the criteria and create a new row that combines those two rows side-by-side. A single row in one of the tables could be joined a single time, or no times, or several times.

Below are a few examples of grouping together the rows of the `cities` table:

```
-- Combine cities & phone_book to map each person to their state
SELECT phone_book.First, phone_book.Last, cities.State
FROM cities
INNER JOIN phone_book
ON cities.Name = phone_book.City
ORDER BY cities.State;

-- Combines cities & phone_book to count the number of people per state
SELECT cities.State, Count(*) AS Num_People
FROM cities
INNER JOIN phone_book
ON cities.Name = phone_book.city
GROUP BY cities.State;
```

Joins allow us to combine everything we've learned so far into far more complex queries. The `demotables.db` database contains 2 more tables: `foods` and `orders`, which you can also explore. These are used to demonstrate a couple more complex queries such as:

- Counting the number of food orders per city
- Adding up how many pounds of each food item were ordered
- Adding up how many pounds were ordered per state
- Counting how many times a specific food item was ordered in each state

These combinations include all 4 of the tables in the database, sometimes all at once.