# CS 122: SQL Lab
**Due: March 7th at 6pm**

You should work alone on this lab, which is designed to give you experience with writing SQL queries. Your task is to do the exercises in Sections 2 through 6. To get started run `svn up` to pick up the materials for this lab (`week8`). To submit your solutions, cut and paste your query and the results obtained for each exercise to the file "exercises.txt." Remember to check in your solution!

## 1 SQLite

In this lab, we will be using the SQLite relational DBMS (`http://sqlite.org/`). Using this light-weight DBMS will allow us to experiment with SQL and get a feel for how a database works, without having to go through the more complex interactions involved in using heavier-weight systems like MySQL, PostgreSQL, Oracle, etc.

One of SQLite's main advantages is that it stores a database in a single file which can be easily copied and moved around, and accessed directly through a command-line interface or simple APIs. Larger DBMSs also store their data in files, but they are rarely (if ever) manipulated directly by the user/programmer. Instead, they must be accessed through a database server, which is typically not easy to set up. On the other hand, SQLite is simple to install and requires no configuration, so you can start toying and tinkering with it right away, and using an existing database is as easy as making a copy of a database file. More details about SQLite's main features can be found at `http://www.sqlite.org/different.html`.

Of course, SQLite does have its drawbacks. Most notably, it does not scale well to high-concurrency applications (e.g., high-traffic websites) or large datasets. Additionally, it does not support all the features of SQL, although this will not be a problem for the simple queries we will be seeing in this lab.

## 2 Using SQLite

Starting SQLite from one of the CS machines is as simple as running the `sqlite3` command with the name of a database:

```
$ sqlite3 planet_express.db
```

This command starts up a SQLite shell where we can type in SQL statements and loads the pre-existing database from the file `planet_express.db`. Besides SQL, the SQLite shell also provides several administration commands, which all begin with a dot. Two commands we will find useful are `.tables` and `.schema`. The first one will show a list of all the tables in the database:

```
sqlite> .tables
Client         Has_Clearance  Planet
Employee       Package        Shipment
```

The `.schema` command shows the DDL (Data Definition Language) command that was used to create a table:

```
sqlite> .schema Planet
CREATE TABLE Planet (
        PlanetID INTEGER PRIMARY KEY NOT NULL,
        Name TEXT NOT NULL,
        Coordinates REAL NOT NULL
);
```

Running `.schema` without any parameter will dump the entire schema for the database.

Now, let's try to run an SQL query that returns the entire contents of the `Planet` table. Type in the following:

```
sqlite> SELECT * FROM Planet;
```

You should see the following output:

```
1|Omicron Persei 8|89475345.3545
2|Decapod X|65498463216.3466
3|Mars|32435021.65468
4|Omega III|98432121.5464
5|Tarantulon VI|849842198.354654
6|Cannibalon|654321987.21654
7|DogDoo VII|65498721354.688
8|Nintenduu 64|6543219894.1654
9|Amazonia|65432135979.6547
```

By default, SQLite makes no attempt to pretty-print the results of a query (this is good to process the output of a query with another program, but not that good if it's going to be read by a human). You can make the output look nicer by running the following administration commands:

```
sqlite> .mode column
sqlite> .headers on
```

If you rerun the previous query, you should now see the following:

```
PlanetID    Name              Coordinates
----------  ----------------  -------------
1           Omicron Persei 8  89475345.3545
2           Decapod X         65498463216.3
3           Mars              32435021.6546
4           Omega III         98432121.5464
5           Tarantulon VI     849842198.354
6           Cannibalon        654321987.216
7           DogDoo VII        65498721354.6
8           Nintenduu 64      6543219894.16
9           Amazonia          65432135979.6
```

To avoid typing this every time you start an interactive session, you can run SQLite with the following arguments:

```
$ sqlite3 -column -header planet_express.db
```

Finally, note that you can break a query into as many lines as you want. The end of a query is always delimited by a semicolon. For example:

```
sqlite> SELECT *
    ..> FROM Planet;
```

# 3 Selection and Projection

SQL provides operations for selecting rows based on specific conditions (known as selection) and for selecting columns (known as projection). Here is a simple query that finds employees with salaries greater than or equal to $10,000.

```
sqlite> SELECT * FROM Employee
    ..> WHERE Salary >= 10000;
```

The above query should return the following:

```
EmployeeID  Name            Position      Salary      Remarks
----------  --------------  ------------  ----------  --------------------------
1           Phillip J. Fry  Delivery boy  7500.0      Not to be confused with ...
2           Turanga Leela   Captain       10000.0
3           Bender Bending  Robot         7500.0
4           Hubert J. Farn  CEO           20000.0
5           John A. Zoidbe  Physician     25.0
6           Amy Wong        Intern        5000.0
7           Hermes Conrad   Bureaucrat    10000.0
8           Scruffy Scruff  Janitor       5000.0
```

Here is a simple projection query that extracts the EmployeeID, Name and Position columns from Employee:

```
sqlite> SELECT EmployeeID, Name, Position
   ...> FROM Employee;
```

You should see the following output:

```
EmployeeID  Name            Position
----------  --------------  ------------
1           Phillip J. Fry  Delivery boy
2           Turanga Leela   Captain
3           Bender Bending  Robot
4           Hubert J. Farn  CEO
5           John A. Zoidbe  Physician
6           Amy Wong        Intern
7           Hermes Conrad   Bureaucrat
8           Scruffy Scruff  Janitor
```

Here's an example that combines the two concepts and shows the usage of `AND` to find the names of employees who make between $10,000 and $15,000.

```
sqlite> SELECT Name
   ...> FROM Employee
   ...> WHERE Salary >= 10000 AND Salary <= 15000;
Name
-------------
Turanga Leela
Hermes Conrad
```

SQL also allows you to remove duplicates from the results. For example, the following query generates a list of senders account numbers:

```
sqlite> SELECT DISTINCT Sender
   ...> FROM Package;
Sender
----------
1
2
3
4
5
6
8
9
10
```

The value `NULL` is used to indicate that a row does not have a value for a particular attribute (column). For example, the Date attribute in the Shipment table is `NULL` when a shipment is still pending. We can use the test "`IS NULL`" to determine whether a value is `NULL`. Here's a query that finds all the pending shipments:

```
sqlite> SELECT *
   ...> FROM Shipment
   ...> WHERE Date IS NULL;
ShipmentID  Date        Manager     Planet
----------  ----------  ----------  ----------
3                       2           3
4                       2           4
5                       7           5
```

### 3.1 Exercises

Write queries to generate the requested information. Remember to copy your query and your results to the files "exercises.txt". Hint: Use the `.schema` command to find out columns in each table.

1. List all the packages from shipment 3.
2. List the shipment number and weight of all the packages.
3. List all the packages from shipment 3, with a weight larger than 10.

## 4 Aggregation and group by

We would like to be able to ask questions such as "How many customers does the shipping company have?" or "What is the weight of the heaviest package?" SQL provides a collection of aggregation operations (`COUNT`, `SUM`, `MAX`, `MIN`, `AVG`, etc.) to answer questions like these.

Here's a query that determines the number of customers:

```
sqlite> SELECT COUNT(*)
   ...> FROM Client;
COUNT(*)
----------
11
```

And here's a query that computes the weight of the heaviest package:

```
sqlite> SELECT MAX(Weight)
   ...> FROM Package;
MAX(Weight)
-----------
100.0
```

Often, there are natural groups of rows in a table, for example, the packages in a shipment or the packages sent by each sender. SQL's `GROUP BY` clause is used in combination with the aggregation operations to answer questions about groups. For example, below is a query to compute the number of packages in each shipment. To understand this example better, you might want to print the entire Package table for comparison.

```
sqlite> SELECT Shipment, COUNT(*)
   ...> FROM Package
   ...> GROUP BY Shipment;
Shipment    COUNT(*)
----------  ----------
1           1
2           2
3           3
4           2
5           4
6           2
```

If we would like the list sorted by the number of packages, we can use the `ORDER BY` clause.

```
sqlite> SELECT COUNT(*) AS Count, Shipment
   ...> FROM Package
   ...> GROUP BY Shipment
   ...> ORDER BY Count;
Count       Shipment
----------  ----------
1           1
2           2
2           4
2           6
3           3
4           5
```

Adding `AS Count` after `COUNT(*)` introduces a name for the corresponding column in the output. Notice that this name can be used later in the query to specify a sort key for the output.

You can sort the rows in decreasing order by adding the keyword `DESC`. Also, you can limit the number of rows printed using the `LIMIT` clause. Here's a query that determines which shipment has the most packages:

```
sqlite> SELECT Shipment, COUNT(*) AS Count
   ...> FROM Package
   ...> GROUP BY Shipment
   ...> ORDER BY Count DESC
   ...> LIMIT 1;
Shipment    Count
----------  ----------
5           4
```

### 4.1 Exercise

Write queries to answer the following questions:

1. What is the average weight of a package?
2. What is the the average weight of a package per sender?
3. Who was the sender of the lightest package and what is its weight?

## 5 Crossing Tables

For complex problems we may wish to query across multiple tables. A way to accomplish this task is to combine several tables into a single query. If we were interested in the clearance levels of various employees we can ask questions from both the Employee table and the Has_Clearance table.

```
sqlite> SELECT * FROM Employee JOIN Has_Clearance;
```

This will produce an extremely long output because it considers all possible crossings of the data. We can restrict the output to make more sense by requiring that the Employee.EmployeeID values match the Has_Clearance.Employee values.

```
sqlite> SELECT EmployeeID, Name, Planet, Level
   ...> FROM Employee
   ...> JOIN Has_Clearance ON Employee.EmployeeID = Has_Clearance.Employee;

   EmployeeID  Name            Planet      Level
   ----------  --------------  ----------  ----------
   1           Phillip J. Fry  1           2
   1           Phillip J. Fry  2           3
   2           Turanga Leela   3           2
   2           Turanga Leela   4           4
   3           Bender Bending  5           2
   3           Bender Bending  6           4
   4           Hubert J. Farn  7           1
```

We can then use this operation to perform intelligent queries that span both tables. For example we may wish to create a roster of names for a delivery which requires at least Clearance Level 2.

```
sqlite> SELECT EmployeeID, Name, Planet, Level
   ...> FROM Employee
   ...> JOIN Has_Clearance ON Employee.EmployeeID = Has_Clearance.Employee
   ...> WHERE Has_Clearance.Level >= 2;
   EmployeeID  Name            Planet      Level
   ----------  --------------  ----------  ----------
   1           Phillip J. Fry  1           2
   1           Phillip J. Fry  2           3
   2           Turanga Leela   3           2
   2           Turanga Leela   4           4
   3           Bender Bending  5           2
   3           Bender Bending  6           4
```

### 5.1 Exercises

1. Write a query that prints the account numbers of the Sender and the Recipient of all Packages that are on Shipments under the Supervision of Manager 2.

2. Write a query to determine the account numbers of recipients of packages that are in pending shipments. (Hint: `DISTINCT` will be useful here.)

## 6 Subqueries

SQL allow us to nest queries to answer more complex queries, such, as "Which client sent the heaviest package?"

```
sqlite> SELECT Name
   ...> FROM Client JOIN Package ON
   ...> Client.AccountNumber = Package.Sender
   ...> WHERE Package.Weight = (SELECT MAX(Weight) FROM Package);
Name
----------
Leo Wong
```

or "Which clients have sent packages?"

```
sqlite> SELECT DISTINCT Name
   ...> FROM Client
   ...> WHERE AccountNumber IN (SELECT Sender FROM Package);
Name
--------------
Al Gore's Head
Barbados Slim
John Zoidfarb
Judge John Whi
Leo Wong
Lrrr
Morbo
Ogden Wernstro
Zapp Brannigan
```

### 6.1 Exercises

1. Write a query to determine the name of the planet that is the destination of the most packages?

2. Write a query to determine the names of clients who have received their packages.

## 7 Inserting new rows

Adding new rows is done using the `INSERT` command. For example, let's say we want to add a new planet:

```
sqlite> INSERT INTO Planet (PlanetID, Name, Coordinates)
   ...> VALUES (10, 'Jupiter', 1839102.5);
```

Recall that the primary key of a table is the name of a column or a set of column names that uniquely identifies each row in the table. If you look at the schema for the Planet table, you will see that PlanetID is specified to be the primary key the table. What happens we try to violate the primary key integrity of the Planet table by running the previous `INSERT` again?

**Acknowledgements:** This lab was adapted from Borja Sotomayor's CMSC23500 Databases course.