SQL TUTORIAL

I. INSTALLATION AND SETUP: SQLITE, DBVIS, AND DATASET

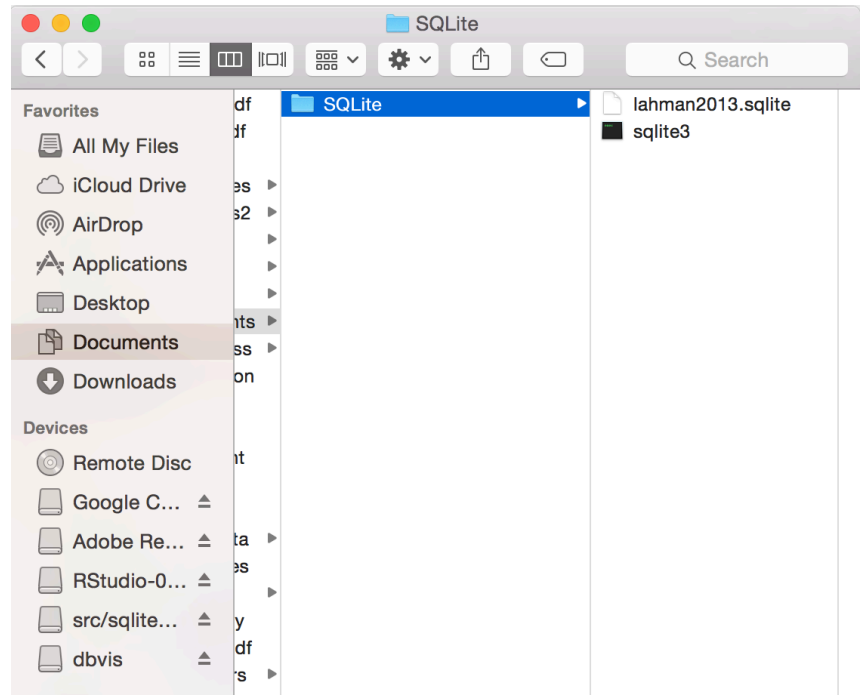
II. SQL SYNTAX

- A. SELECT STATEMENTS, WHERE, ORDER BY
- B. TABLE JOINS, GROUP BY
- C. DISTINCT, CASE, IS NOT NULL
- D. SUB-QUERIES
- E. CREATE TABLE

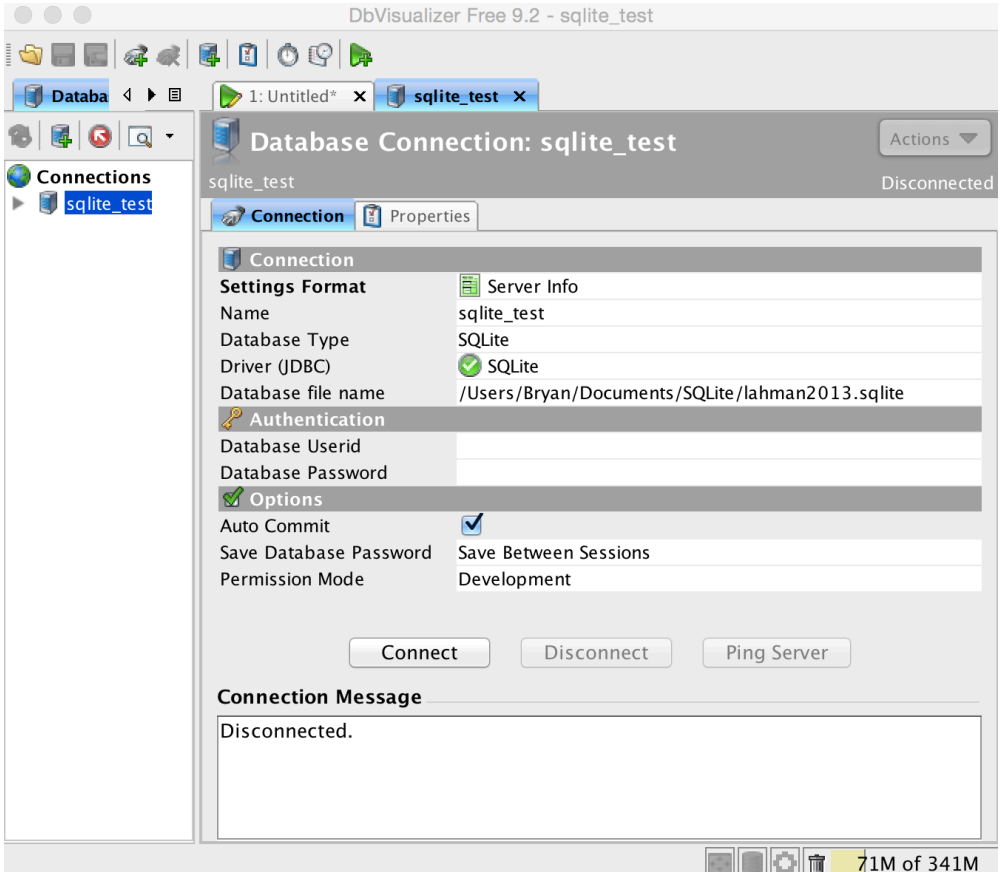
II. USING PANDAS AND SQL SEAMLESSLY

INSTALLATION AND SETUP

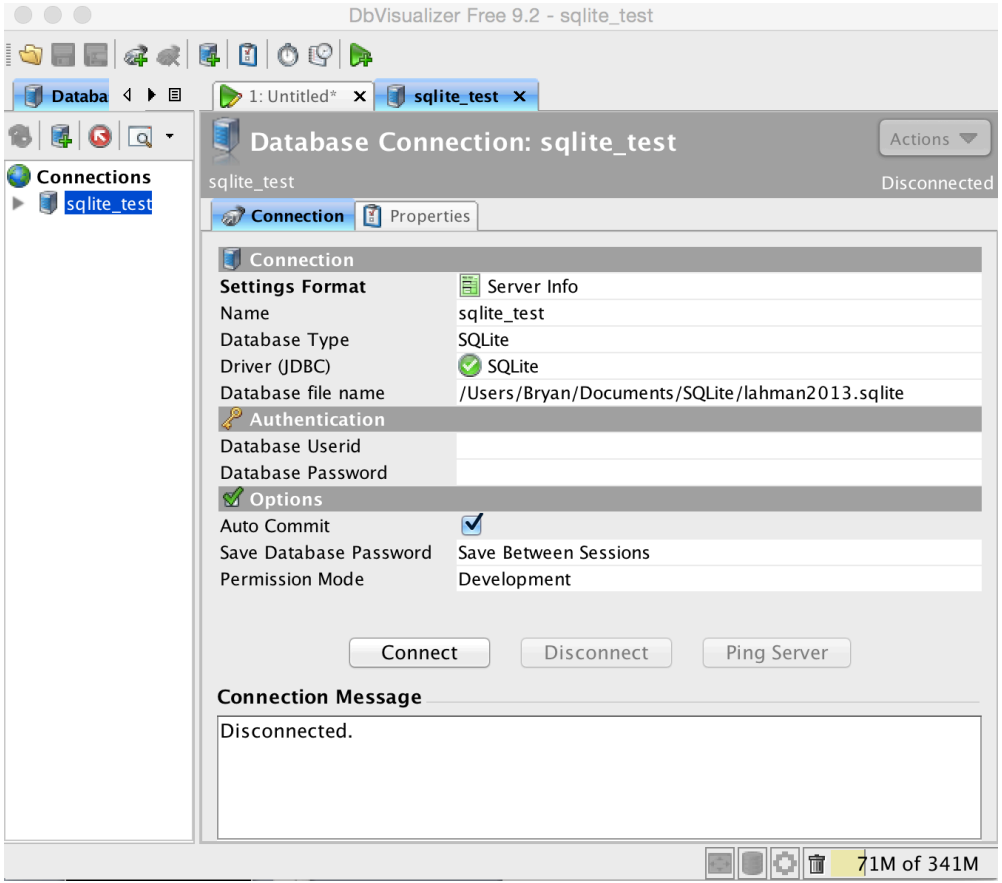
- Create a new folder called ‘SQLite’ in your My Documents folder.
- Unzip SQLite and drag the sqlite3 application into the folder you just created.
- Move the Baseball dataset to the SQLite folder.
- Open Sublime Text Editor.
- Open DBVisualizer.



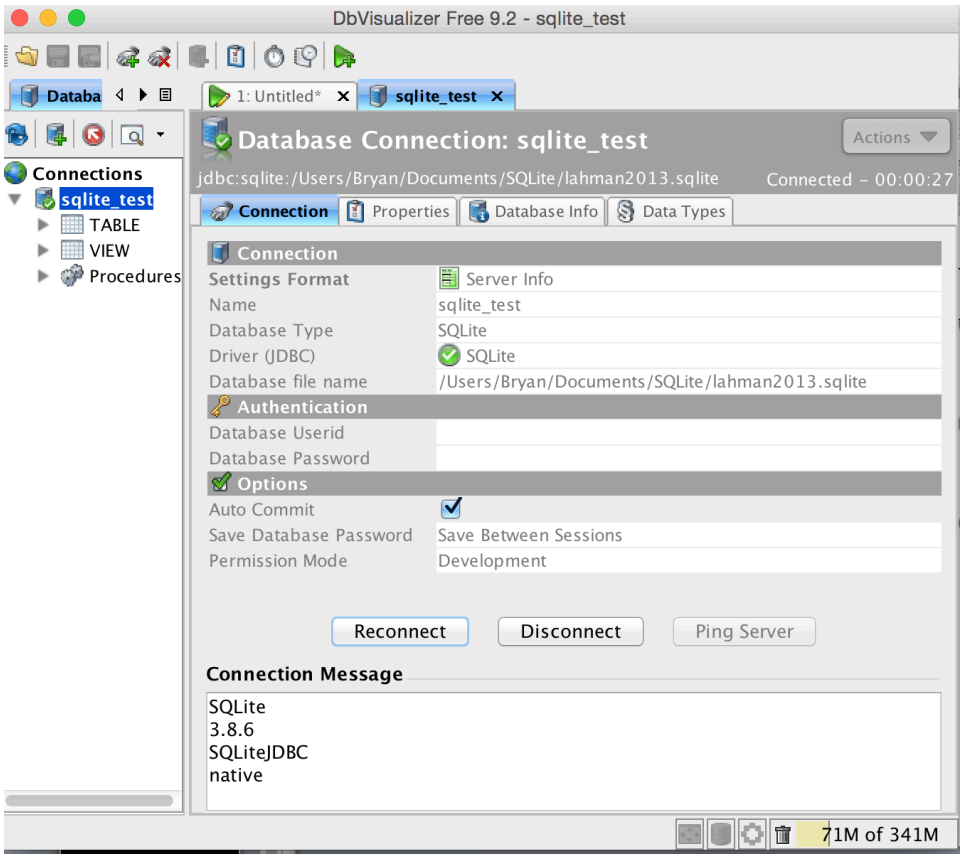
- In DBVisualizer, go to Database -> Create Database Connection.
- Click ‘No Wizard’ in the ‘Use Connection Wizard’ prompt.
- You should see the window to the right appear.



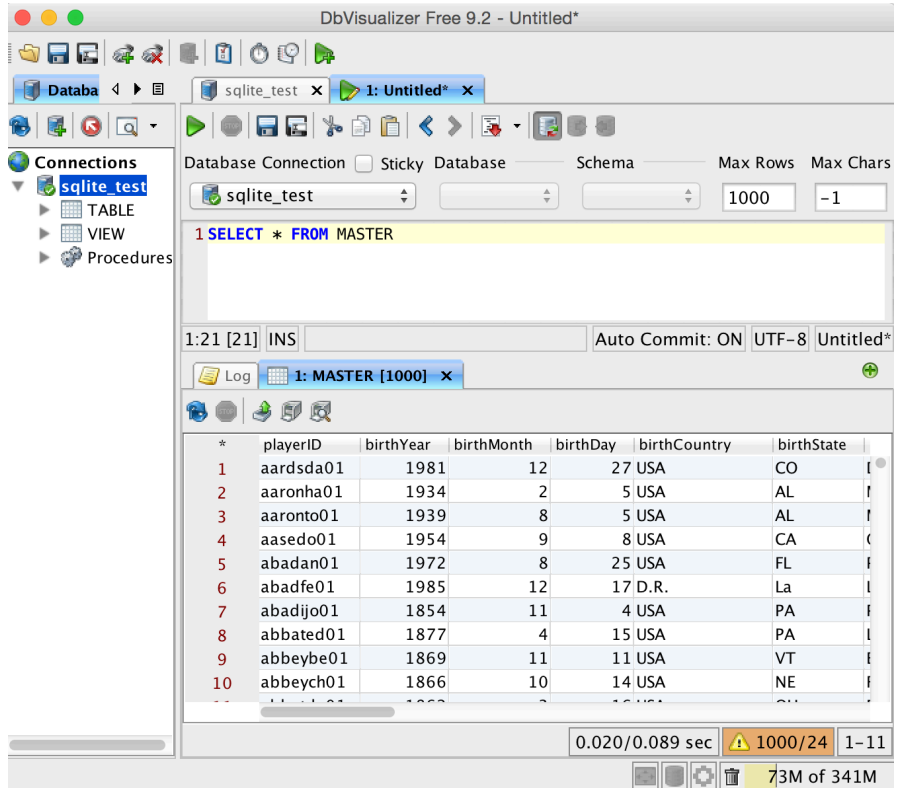
- Under ‘name’, name your database. I suggest ‘baseball_stats’ to start.
- Under ‘Database Type’, select SQLite.
- Under ‘Driver (JDBC)’, select SQLite.
- Insert the path to the baseball dataset file under ‘Database File Name’.



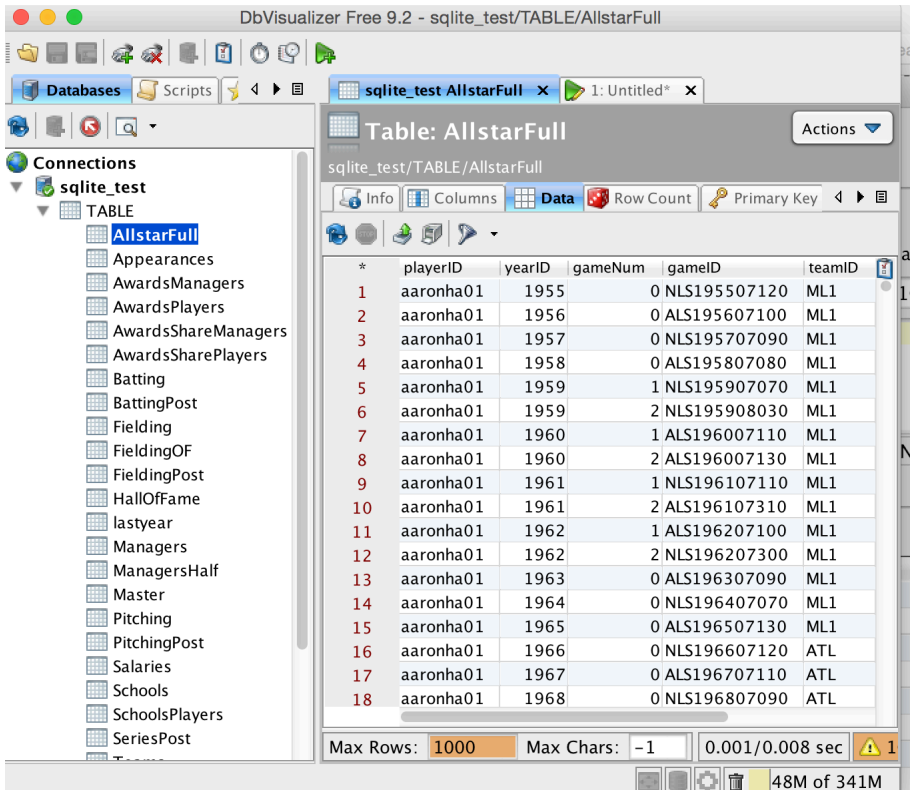
- Do not put in a database user id or password.
- Click ‘Connect’. You should see ‘SQLite’ appear under ‘Connection Message’ and the other options go gray.
- If the connection failed, check the path to the database file – it should not be quoted.



- Go to SQL Commander-> New SQL Commander.
- Type SELECT * FROM Master; and type Command-Enter.
- You've run your first SQL query!
- Pressing the down arrow on TABLE lets you see all the tables in the database.



- Double-click on a table to see its information such as columns, data, and indexes.
- Press the ‘Data’ tab to see what the table looks like.
- Go back to your SQL Commander window so that we can begin writing queries.



- The SELECT statement is basis of most analytic queries you will write. It lets you select all the data in a particular table.

- You designate the name of the table in the FROM clause. All SQL statements should end in a semicolon.

- SELECT * FROM {table}; lets you select ALL the columns in the table.

- SELECT {colname1}, {colname2}, {colname3} FROM {table}; lets you select only the columns you want from the table.

- SELECT {conmane1} as your_name FROM {table) lets you give your own designated alias name to the column in the SQL output.

EXERCISE: SELECT STATEMENTS

1. Select all the data in the table named ‘Master’.
2. Select all the data in the table named ‘Batting’.
3. Select only the columns in ‘Master’ named playerID, nameGiven, and birthYear.
4. Select only the columns in ‘Batting’ named playerID, yearID, and TeamID.
5. In ‘Batting’, select and give the following aliases to lgID, G_batting, AB, R, and H: league_id, games, games_as_batter, at_bats, runs, and hits.

```sql
--1) Simple SELECT statements
SELECT * FROM Master;
SELECT * FROM Batting;
SELECT * FROM Pitching;
SELECT * FROM Fielding;


--2) SELECT statements designating only one column
SELECT playerID, nameGiven, birthYear FROM Master;
SELECT playerID, yearID, TeamID FROM Batting;

--2A) Giving aliases to columns
SELECT playerID, yearID, TeamID, lgID as league_id,
FROM Batting;
--question: at what granularity (by player? by game


--3) Using SELECT and WHERE to limit data
SELECT playerID, yearID, TeamID, lgID as league_id,
AB as at_bats, R as Runs, H as Hits
FROM Batting
WHERE yearID > 2000;

--3a) Multiple WHERE conditions
SELECT playerID, yearID, TeamID, lgID as league_id,
AB as at_bats, R as Runs, H as Hits
FROM Batting
WHERE yearID > 2000 AND playerID = 'aardsda01';

```

- The WHERE clause lets you designate limits to the rows selected by SQL, and goes after FROM in the SELECT statement.
- The syntax is: SELECT {columns} FROM {table} WHERE {condition};.
- The >, <, <=, >=, =, !=, OR, AND, IS/IS NOT NULL, BETWEEN, and IN operands are available for use in the WHERE clause.
- Numbers are supported natively, strings and textual dates must be enclosed in apostrophes.
- Multiple conditions can be added to the statement using AND.
- The in({element1},{element2}) feature lets you select rows where a column value is in a list.

EXERCISE: THE WHERE CLAUSE

1. Open Sublime Text Editor and create a new file named sql_tutorial.sql in your ‘My Documents’ folder. Save your your SQL commander text to the new file.

2. Select all rows in ‘Batting’ with yearID greater than 2000.

3. Select all rows in ‘Batting’ with yearID greater than 2000 and with the playerID of ‘aardsda01’.

4. Select all rows in ‘Batting’ with yearID greater than 2000 and with the playerIDs in a list containing 'aardsda01’ and 'abbotpa01’.

```
18  --3) Using SELECT and WHERE to limit data
19  SELECT playerID, yearID, TeamID, lgID as league_id, G as games, G_ba
20  AB as at_bats, R as Runs, H as Hits
21  FROM Batting
22  WHERE yearID > 2000;
23
24  --3a) Multiple WHERE conditions
25  SELECT playerID, yearID, TeamID, lgID as league_id, G as games, G_ba
26  AB as at_bats, R as Runs, H as Hits
27  FROM Batting
28  WHERE yearID > 2000 AND playerID = 'aardsda01';
29
30  --3b) Listed WHERE conditions
31  SELECT playerID, yearID, TeamID, lgID as league_id, G as games, G_ba
32  AB as at_bats, R as Runs, H as Hits
33  FROM Batting
34  WHERE yearID > 2000 AND playerID in( 'aardsda01', 'abbotpa01');
35
36
37  --4) ORDER BY (ASC or DESC)
38  SELECT playerID, yearID, TeamID, lgID as league_id, G as games, G_ba
39  AB as at_bats, R as Runs, H as Hits
40  FROM Batting
41  WHERE yearID > 2000 AND playerID = 'aardsda01'
42  ORDER BY games_as_batter DESC;
43  --Quesion: what are the years where this player had the most number
44
45  --4a) ORDER BY two columns
46  --say we want to know games at batter by league
47  SELECT playerID, yearID, TeamID, lgID as league_id, G as games, G_ba
48  AB as at_bats, R as Runs, H as Hits
49  FROM Batting
50  WHERE yearID > 2000 AND playerID = 'aardsda01'
51  ORDER BY league_id ASC, games_as_batter DESC;
52
53
```

- ORDER BY lets you order the data returned by the SELECT clause according to one or more columns.
- The syntax is: SELECT {columns} FROM {table} WHERE {conditions} ORDER BY {column};
- Put ASC or DESC after the column name to designate whether you want the data in descending order (highest to lowest) or ascending order (lowest to highest).
- You can order by multiple columns with the following syntax: ORDER BY {column1} (ASC/DESC), {column2} (ASC/DESC).
- ORDER BY works for most data types you will encounter, including numeric, string, and date data.

EXERCISE: ORDER BY

1. Save your last query to Sublime.
2. Alter your last query to be ordered by games_as_batter in descending order.
3. What are the years where this player had the most number of games at bat?
4. Add another column to ORDER BY so that we first order by league ID in ascending order, and then in games at bat in descending order.

```
18  --3) Using SELECT and WHERE to limit data
19  SELECT playerID, yearID, TeamID, lgID as league_id, G as games, G_ba
20  AB as at_bats, R as Runs, H as Hits
21  FROM Batting
22  WHERE yearID > 2000;
23
24  --3a) Multiple WHERE conditions
25  SELECT playerID, yearID, TeamID, lgID as league_id, G as games, G_ba
26  AB as at_bats, R as Runs, H as Hits
27  FROM Batting
28  WHERE yearID > 2000 AND playerID = 'aardsda01';
29
30  --3b) Listed WHERE conditions
31  SELECT playerID, yearID, TeamID, lgID as league_id, G as games, G_ba
32  AB as at_bats, R as Runs, H as Hits
33  FROM Batting
34  WHERE yearID > 2000 AND playerID in( 'aardsda01', 'abbotpa01');
35
36
37  --4) ORDER BY (ASC or DESC)
38  SELECT playerID, yearID, TeamID, lgID as league_id, G as games, G_ba
39  AB as at_bats, R as Runs, H as Hits
40  FROM Batting
41  WHERE yearID > 2000 AND playerID = 'aardsda01'
42  ORDER BY games_as_batter DESC;
43  --Quesion: what are the years where this player had the most number
44
45  --4a) ORDER BY two columns
46  --say we want to know games at batter by league
47  SELECT playerID, yearID, TeamID, lgID as league_id, G as games, G_ba
48  AB as at_bats, R as Runs, H as Hits
49  FROM Batting
50  WHERE yearID > 2000 AND playerID = 'aardsda01'
51  ORDER BY league_id ASC, games_as_batter DESC;
52
53
```

- Table joins return different tables joined together by a common index.
- The syntax is: SELECT t.{columns}, t2.{columns} FROM {table} t (LEFT / RIGHT / INNER) JOIN {table} t2 ON {common index};
- LEFT JOIN returns the first table you designate in full, and has NULL values for those rows that do not have a corresponding index in the second table. INNER JOIN returns only those rows with a common index, all rows that don't have one are not returned.
- As the two tables may share column names, we add table aliases (ie t2.column and t1.column in both the SELECT and JOIN clauses. We designated the aliases at the start of the JOIN clause.
- Be aware of many-on-one joins, and one-on-many joins if the index is non-unique in one or both of the columns.

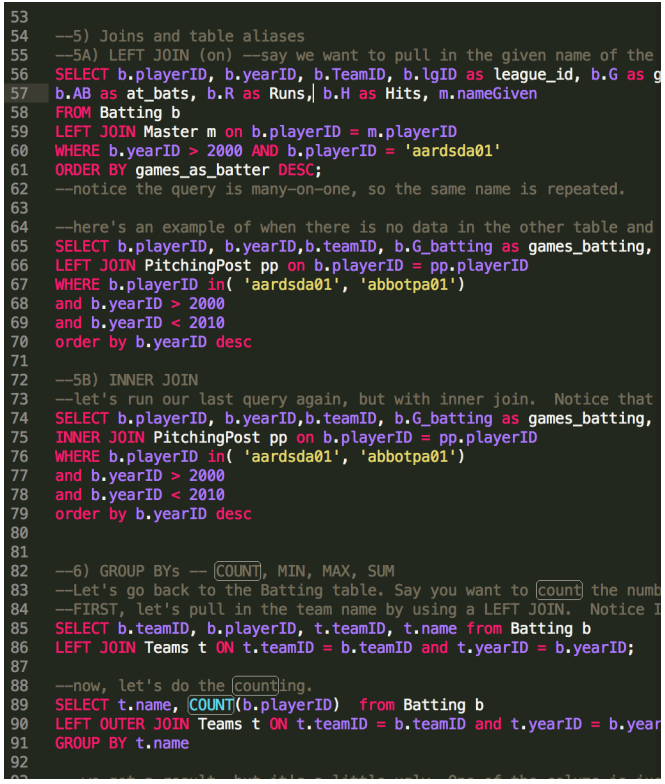
EXERCISE: JOINS

1. Save your last query to Sublime.

2. LEFT JOIN ‘Batting’ to Master on Batting.playerID = Master.playerID

3. Only select the nameGiven column from Master and relevant statistics (year, team, league, games, games_as_batter, at_bats, hits, and runs) from Batting so that we can get a concise view of a player’s profile. Remember to use table aliases!

4. Repeat the same process with an INNER JOIN.



```
53
54  --5) Joins and table aliases
55  --5A) LEFT JOIN (on) --say we want to pull in the given name of the
56  SELECT b.playerID, b.yearID, b.TeamID, b.lgID as league_id, b.G as g
57  b.AB as at_bats, b.R as Runs, b.H as Hits, m.nameGiven
58  FROM Batting b
59  LEFT JOIN Master m on b.playerID = m.playerID
60  WHERE b.yearID > 2000 AND b.playerID = 'aardsda01'
61  ORDER BY games_as_batter DESC;
62  --notice the query is many-on-one, so the same name is repeated.
63
64  --here's an example of when there is no data in the other table and
65  SELECT b.playerID, b.yearID,b.teamID, b.G_batting as games_batting,
66  LEFT JOIN PitchingPost pp on b.playerID = pp.playerID
67  WHERE b.playerID in( 'aardsda01', 'abbotpa01')
68  and b.yearID > 2000
69  and b.yearID < 2010
70  order by b.yearID desc
71
72  --5B) INNER JOIN
73  --let's run our last query again, but with inner join.  Notice that
74  SELECT b.playerID, b.yearID,b.teamID, b.G_batting as games_batting,
75  INNER JOIN PitchingPost pp on b.playerID = pp.playerID
76  WHERE b.playerID in( 'aardsda01', 'abbotpa01')
77  and b.yearID > 2000
78  and b.yearID < 2010
79  order by b.yearID desc
80
81
82  --6) GROUP BYs -- COUNT, MIN, MAX, SUM
83  --Let's go back to the Batting table. Say you want to count the numb
84  --FIRST, let's pull in the team name by using a LEFT JOIN.  Notice I
85  SELECT b.teamID, b.playerID, t.teamID, t.name from Batting b
86  LEFT JOIN Teams t ON t.teamID = b.teamID and t.yearID = b.yearID;
87
88  --now, let's do the counting.
89  SELECT t.name, COUNT(b.playerID)  from Batting b
90  LEFT OUTER JOIN Teams t ON t.teamID = b.teamID and t.yearID = b.year
91  GROUP BY t.name
92
```

- The GROUP BY clause aggregates data by a column dimension.
- The syntax is SELECT {column1} (COUNT/MIN/MAX/AVG/SUM) ({column2}) WHERE {conditions} GROUP BY {column1};
- GROUP BY lets you answer questions like – how many players are there per team? What's the average number of at-bats for a player? What's the total number of at-bats for a player?
- You can GROUP BY more than one table column to get more granular aggregation. Ex: what's the average number of at-bats for a player at each team they played on.
- Combine GROUP BY with ORDER BY to rank aggregate statistics. Ex: what are the players with the top number of at-bats per year?

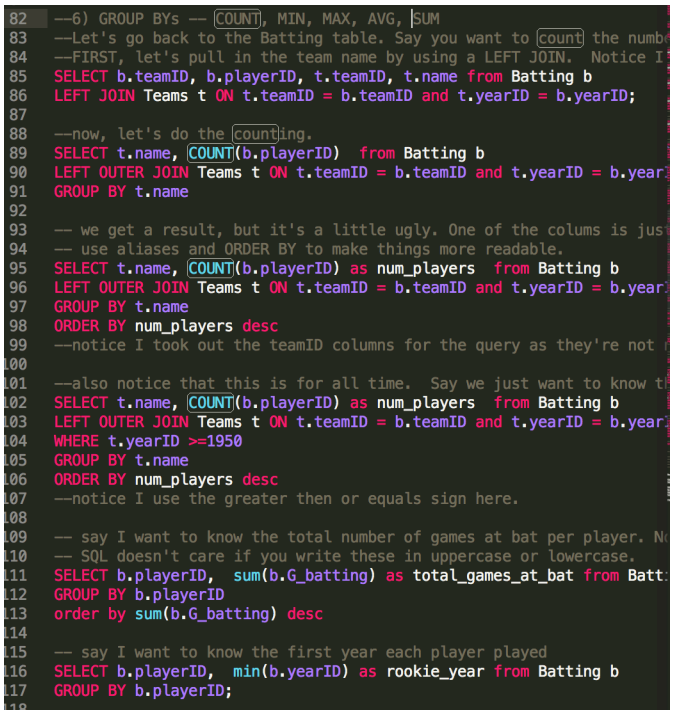
EXERCISE: GROUP BY

1. LEFT JOIN the Batting table to the Teams table on two common indices: teamID and yearID.

2. Count the number of player IDs per team name. Enhance your query by ordering the output by the number of players per team. Limit your query to data on or after 1950.

3. Find the top players by total number of games at bat in the Batting table.

4. Get the rookie year (ie the minimum year) of every player in the Batting table.



```
--6) GROUP BYs -- COUNT, MIN, MAX, AVG, SUM
--Let's go back to the Batting table. Say you want to count the numb
--FIRST, let's pull in the team name by using a LEFT JOIN.  Notice I
SELECT b.teamID, b.playerID, t.teamID, t.name from Batting b
LEFT JOIN Teams t ON t.teamID = b.teamID and t.yearID = b.yearID;

--now, let's do the counting.
SELECT t.name, COUNT(b.playerID)  from Batting b
LEFT OUTER JOIN Teams t ON t.teamID = b.teamID and t.yearID = b.year
GROUP BY t.name

-- we get a result, but it's a little ugly. One of the colums is jus
-- use aliases and ORDER BY to make things more readable.
SELECT t.name, COUNT(b.playerID) as num_players  from Batting b
LEFT OUTER JOIN Teams t ON t.teamID = b.teamID and t.yearID = b.year
GROUP BY t.name
ORDER BY num_players desc
--notice I took out the teamID columns for the query as they're not

--also notice that this is for all time.  Say we just want to know t
SELECT t.name, COUNT(b.playerID) as num_players  from Batting b
LEFT OUTER JOIN Teams t ON t.teamID = b.teamID and t.yearID = b.year
WHERE t.yearID >=1950
GROUP BY t.name
ORDER BY num_players desc
--notice I use the greater then or equals sign here.

-- say I want to know the total number of games at bat per player. N
-- SQL doesn't care if you write these in uppercase or lowercase.
SELECT b.playerID,  sum(b.G_batting) as total_games_at_bat from Batt
GROUP BY b.playerID
order by sum(b.G_batting) desc

-- say I want to know the first year each player played
SELECT b.playerID,  min(b.yearID) as rookie_year from Batting b
GROUP BY b.playerID;
```

- Adding DISTINCT to the SELECT statement returns only unique entries in a column.
- The syntax is SELECT DISTINCT {colname} FROM {table};
- DISTINCT is useful when used with GROUP BY. Ex: what are the number of unique player IDs in the Batting table?
- CASE statements let you create a new column based on conditions in another column.
- The syntax is SELECT CASE WHEN {colname} = {condition} THEN {output when true} ELSE {output when false} END FROM {table};
- This is useful when you want to create a new label for your data, such as flagging players with a large number of at-bats.

EXERCISE: DISTINCT AND CASE

1. SELECT the distinct player IDs from the Batting table.

2. COUNT the number of distinct IDs, and then count all the IDs. Look at the difference in the COUNT.

3. Create a new column called many_games_batted using CASE that has a 1 if a player batted 20 or more games in a season and 0 if a player batted fewer than 20 games.

```
--7) DISTINCT
--Notice that playerID repeats in the Batting table.  Say you want to
SELECT DISTINCT playerID FROM Batting;
--Then, say you want know how many unique player IDs exist in the ta
SELECT COUNT(DISTINCT playerID) FROM Batting;
--Compare this to the number of all (unique and non-unique player ID
SELECT COUNT(playerID) FROM Batting;
--there are many, many more nondistinct entries than distinct ones.


--8) CASE statements
--Say you want to make a new label for players who had 20 or more ga
SELECT CASE WHEN b.G_batting >=20 THEN 1 ELSE 0 END as many_games_ba
FROM Batting b;
--notice my use of b.* to select all columns without explicitly nami


--9) Subqueries and IS NOT NULL
-- say you wanted to know the batting information for each player's
-- first, find the last year of each player in the Batting table --
SELECT  playerID, max(yearID) as maxyear from Batting
GROUP BY playerID;

--then, we throw it into a subquery
select sq1.maxyear, b.* from Batting b
LEFT OUTER JOIN
(SELECT  playerID, max(yearID) as maxyear from Batting
GROUP BY playerID) sq1
ON b.playerID = sq1.playerID
AND b.yearID = sq1.maxyear;

-- now I want to only select situations where maxyear is not null --
-- so, I add to the WHERE clause
select sq1.maxyear, b.* from Batting b
LEFT OUTER JOIN
(SELECT  playerID, max(yearID) as maxyear from Batting
GROUP BY playerID) sq1
ON b.playerID = sq1.playerID
AND b.yearID = sq1.maxyear
WHERE  sq1.maxyear is not null;
```

- Subqueries are a very helpful way to join information you've already grouped or altered to an existing or other altered table.
- Example: What was the batting information for a player's last year?
- The syntax is: SELECT {t1.columns}, {t2.columns} FROM {table1} t1 LEFT OUTER JOIN (SELECT {columns} FROM {table2}) t2 ON t1.{joint key} = t2.{joint key};
- In effect, your subquery SQL statement is treated like just another table in the outer SQL statement.

EXERCISE: SUBQUERIES

1. Find the last year of each player in the Batting table using MAX() and GROUP BY.

2. Move that SQL into a sub-query and left join it back to Batting on the two common keys: playerID and year.

3. Add a condition in the WHERE clause so that we only return rows where the year in the subquery is not null. This shows the batting information only for each player's last year.

```
--9) Subqueries and IS NOT NULL
-- say you wanted to know the batting information for each player's
-- first, find the last year of each player in the Batting table --
SELECT  playerID, max(yearID) as maxyear from Batting
GROUP BY playerID;

--then, we throw it into a subquery
select sq1.maxyear, b.* from Batting b
LEFT OUTER JOIN
(SELECT  playerID, max(yearID) as maxyear from Batting
GROUP BY playerID) sq1
ON b.playerID = sq1.playerID
AND b.yearID = sq1.maxyear;

-- now I want to only select situations where maxyear is not null --
-- so, I add to the WHERE clause
select sq1.maxyear, b.* from Batting b
LEFT OUTER JOIN
(SELECT  playerID, max(yearID) as maxyear from Batting
GROUP BY playerID) sq1
ON b.playerID = sq1.playerID
AND b.yearID = sq1.maxyear
WHERE  sq1.maxyear is not null;


--10) CREATE TABLE
--say I want to save the max_year data to a new table.  I use the fo
CREATE TABLE lastyear as select sq1.maxyear, b.* from Batting b
LEFT OUTER JOIN
(SELECT  playerID, max(yearID) as maxyear from Batting
GROUP BY playerID) sq1
ON b.playerID = sq1.playerID
AND b.yearID = sq1.maxyear
WHERE  sq1.maxyear is not null;

-- once this is done, I can query from the new table as you do with
SELECT * from lastyear;
```

- New tables in the SQL database can be made using the CREATE TABLE syntax.
- For this lesson, we will only create tables from SELECT statements, but be aware that you can also create empty tables and add new rows using INSERT INTO.
- The syntax for creating a table is: CREATE TABLE {new table name} AS {SELECT statement};
- Once the table is created, you can access it via regular SELECT statemenets.

EXERCISE: CREATE TABLE

1. Create a table of the results of the last subquery you created.

2. Query your new table using a SELECT statement.

```
--9) Subqueries and IS NOT NULL
-- say you wanted to know the batting information for each player's
-- first, find the last year of each player in the Batting table --
SELECT  playerID, max(yearID) as maxyear from Batting
GROUP BY playerID;

--then, we throw it into a subquery
select sq1.maxyear, b.* from Batting b
LEFT OUTER JOIN
(SELECT  playerID, max(yearID) as maxyear from Batting
GROUP BY playerID) sq1
ON b.playerID = sq1.playerID
AND b.yearID = sq1.maxyear;

-- now I want to only select situations where maxyear is not null --
-- so, I add to the WHERE clause
select sq1.maxyear, b.* from Batting b
LEFT OUTER JOIN
(SELECT  playerID, max(yearID) as maxyear from Batting
GROUP BY playerID) sq1
ON b.playerID = sq1.playerID
AND b.yearID = sq1.maxyear
WHERE  sq1.maxyear is not null;


--10) CREATE TABLE
--say I want to save the max_year data to a new table.  I use the fo
CREATE TABLE lastyear as select sq1.maxyear, b.* from Batting b
LEFT OUTER JOIN
(SELECT  playerID, max(yearID) as maxyear from Batting
GROUP BY playerID) sq1
ON b.playerID = sq1.playerID
AND b.yearID = sq1.maxyear
WHERE  sq1.maxyear is not null;

-- once this is done, I can query from the new table as you do with
SELECT * from lastyear;
```

- Pandas interfaces with SQL via its pandas.read_sql() method.
- With read_sql(), you pass a connection (made via the sqlite3 package) and a string containing your SQL.
- read_sql() returns a dataframe with the results of your query.
- Pandas.DataFrame.to_sql() writes a DataFrame back to your SQL database.
- The if_exists option lets you designate what happens if the table you're asking to create already exists. You can either replace the table with the DataFrame you're writing, append to the existing table, or have the function fail.

EXERCISE: PANDAS AND SQL

1. Import sqlite3 and connect to your SQLite database.
2. Pass the SQL from your last in-class exercise to a string.
3. Use the pandas.read_sql function to retrieve the results of your SQL into a pandas DataFrame.
4. Fill the DataFrame's NANs with zero.
5. Write the DataFrame back to your database in a table called 'pandas_table'.

```
import sqlite3
import pandas

# connect to the baseball database. Notice I am passing the full path
# to the SQLite file.
conn = sqlite3.connect('/Users/Bryan/Documents/SQLite/lahman2013.sqlite')

# creating an object contraining a string that has the SQL query. Notice that
# I am using triple quotes to allow my query to exist on multiple lines.
sql = """select sq1.maxyear, b.* from Batting b
LEFT OUTER JOIN
(SELECT  playerID, max(yearID) as maxyear from Batting
GROUP BY playerID) sq1
ON b.playerID = sq1.playerID
AND b.yearID = sq1.maxyear
WHERE  sq1.maxyear is not null"""

# passing the connection and the SQL string to pandas.read_sql.
df = pandas.read_sql(sql, conn)
# NOTE: I can use this syntax for SQLite, but for other flavors of SQL
# (MySQL, PostgreSQL, etc.) you will have to create a SQLAlchemy engine
# as the connection. More information on SQLAlchemy at http://www.sqlalchemy.org/.
# Stack Overflow also has some nice examples of how to make this connection.

# closing the connection.
conn.close()

# filling NaNs
df.fillna(0, inplace = True)

# re-opening the connection to SQLite.
conn = sqlite3.connect('/Users/Bryan/Documents/SQLite/lahman2013.sqlite')
# writing the table back to the database.
# If the table already exists, I'm opting to replace it.
df.to_sql('pandas_table', conn, if_exists = 'replace')
# You can also append to the table if it exists
# with the option if_exists = 'append.'

# closing the connection.
conn.close()
```

1. Find the player with the most at-bats in a single season.
2. Find the name of the the player with the most at-bats in baseball history.
3. Find the average number of at_bats of players in their rookie season.
4. Find the average number of at_bats of players in their final season for all players born after 1980.
5. Find the average number of at_bats of Yankees players who began their second season at or after 1980.
6. Pass #5 into a pandas DataFrame and write it back to SQLite.

QUESTIONS?