# SQL and Aggregation

## SQL Basics

### Creating Tables

You can create SQL tables either from scratch or from existing tables.

The following statement creates a table by specifying column names and values without referencing another table. Each `SELECT` clause specifies the values for one row, and `UNION` is used to join rows together. The `AS` clauses give a name to each column; it need not be repeated in subsequent rows after the first.

```
CREATE TABLE [table_name] AS
  SELECT [val1] AS [column1], [val2] AS [column2], ... UNION
  SELECT [val3]             , [val4]             , ... UNION
  SELECT [val5]             , [val6]             , ...;
```

Let's say we want to make the following table called `big_game` which records the scores for the Big Game each year. This table has three columns: `berkeley`, `stanford`, and `year`.

We could do so with the following `CREATE TABLE` statement:

```
CREATE TABLE big_game AS
  SELECT 30 AS berkeley, 7 AS stanford, 2002 AS year UNION
  SELECT 28,             16,            2003         UNION
  SELECT 17,             38,            2014;
```

### Selecting From Tables

More commonly, we will create new tables by selecting specific columns that we want from existing tables by using a `SELECT` statement as follows:

```
SELECT [columns] FROM [tables] WHERE [condition] ORDER BY [columns] LIMIT [limit];
```

Let's break down this statement:

- `SELECT [columns]` tells SQL that we want to include the given columns in our output table; `[columns]` is a comma-separated list of column names, and `*` can be used to select all columns
- `FROM [table]` tells SQL that the columns we want to select are from the given table
- `WHERE [condition]` filters the output table by only including rows whose values satisfy the given `[condition]`, a boolean expression
- `ORDER BY [columns]` orders the rows in the output table by the given comma-separated list of columns; by default, values are sorted in ascending order (ASC), but you can use DESC to sort in descending order
- `LIMIT [limit]` limits the number of rows in the output table by the integer `[limit]`

Here are some examples:

Select all of Berkeley's scores from the `big_game` table, but only include scores from years past 2002:

```
sqlite> SELECT berkeley FROM big_game WHERE year > 2002;
28
17
```

Select the scores for both schools in years that Berkeley won:

```
sqlite> SELECT berkeley, stanford FROM big_game WHERE berkeley > stanford;
30|7
28|16
```

Select the years that Stanford scored more than 15 points:

```
sqlite> SELECT year FROM big_game WHERE stanford > 15;
2003
2014
```

### SQL operators

Expressions in the `SELECT`, `WHERE`, and `ORDER BY` clauses can contain one or more of the following operators:

- comparison operators: `=`, `>`, `<`, `<=`, `>=`, `<>` or `!=` ("not equal")
- boolean operators: `AND`, `OR`
- arithmetic operators: `+`, `-`, `*`, `/`
- concatenation operator: `||`

Output the ratio of Berkeley's score to Stanford's score each year:

```
sqlite> select berkeley * 1.0 / stanford from big_game;
0.447368421052632
1.75
4.28571428571429
```

Output the sum of scores in years where both teams scored over 10 points:

```
sqlite> select berkeley + stanford from big_game where berkeley > 10 and stanford > 10;
55
44
```

Output a table with a single column and single row containing the value "hello world":

```
sqlite> SELECT "hello" || " " || "world";
hello world
```

*Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.*

# SQL Aggregation

Previously, we have been dealing with queries that process one row at a time. When we join, we make pairwise combinations of all of the rows. When we use `WHERE`, we filter out certain rows based on the condition. Alternatively, applying an aggregate function such as `MAX(column)` combines the values in multiple rows.

By default, we combine the values of the *entire* table. For example, if we wanted to count the number of flights from our `flights` table, we could use:

```
sqlite> SELECT COUNT(*) from FLIGHTS;
13
```

What if we wanted to group together the values in similar rows and perform the aggregation operations within those groups? We use a `GROUP BY` clause.

Here's another example. For each unique departure, collect all of the rows having the same departure airport into a group. Then, select the `price` column and apply the `MIN` aggregation to recover the price of the cheapest departure from that group. The end result is a table of departure airports and the cheapest departing flight.

```
sqlite> SELECT departure, MIN(price) FROM flights GROUP BY departure;
AUH|932
LAS|50
LAX|89
SEA|32
SFO|40
SLC|42
```

Just like how we can filter out rows with `WHERE`, we can also filter out groups with `HAVING`. Typically, a `HAVING` clause should use an aggregation function. Suppose we want to see all airports with at least two departures:

```
sqlite> SELECT departure FROM flights GROUP BY departure HAVING COUNT(*) >= 2;
LAX
SFO
SLC
```

Note that the `COUNT(*)` aggregate just counts the number of rows in each group. Say we want to count the number of *distinct* airports instead. Then, we could use the following query:

```
sqlite> SELECT COUNT(DISTINCT departure) FROM flights;
6
```

This enumerates all the different departure airports available in our `flights` table (in this case: SFO, LAX, AUH, SLC, SEA, and LAS).

# Cities

In this discussion, we will be writing SQL queries on a database containing information on selected cities and states. The data is not guaranteed to be precise or accurate. (In fact, it was obtained by a single TA quickly looking up facts on Wikipedia.)

There are two main tables that you will be querying.

- `cities`: Selected US cities
- `states`: Corresponding states of the select US cities

```
create table cities as
    select 'Berkeley' as name, 'CA' as state, 12000 as population, 1878 as founded, 18.0
    as area union
    select 'San Francisco'   , 'CA'         , 871000            , 1850           , 231.0
     union
    select 'Los Angeles'     , 'CA'         , 3971000           , 1850           , 503.0
     union
    select 'Seattle'         , 'WA'         , 609000            , 1869           , 143.0
     union
    select 'Houston'         , 'TX'         , 2099451           , 1837           , 667.0
     union
    select 'New York City'   , 'NY'         , 8550000           , 1624           , 468.0
     union
    select 'Chicago'         , 'IL'         , 2696000           , 1833           , 234.0
     union
    select 'Philadelphia'    , 'PA'         , 1567000           , 1701           , 142.0
     union
    select 'Phoenix'         , 'AZ'         , 1446000           , 1881           , 518.0
     union
    select 'San Antonio'     , 'TX'         , 1437000           , 1837           , 465.0
     union
    select 'Dallas'          , 'TX'         , 1300000           , 1856           , 386.0
     union
    select 'Jacksonville'    , 'FL'         , 822000            , 1832           ,
    875.0;

create table states as
    select 'California' as name, 'CA' as abbreviation, 39250000.0 as population union
    select 'Washington'        , 'WA'                , 7288000.0 union
    select 'Texas'             , 'TX'                , 27863000.0 union
    select 'New York'          , 'NY'                , 19795000.0 union
    select 'Illinois'          , 'IL'                , 12801000.0 union
    select 'Pennsylvania'      , 'PA'                , 12802503.0 union
    select 'Arizona'           , 'AZ'                , 6828000.0 union
    select 'Florida'           , 'FL'                , 20612000.0;
```

**Q1: California**

Write a query that selects all records for cities in California.

```
    create table california as
        SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

You should get the following output:

```
sqlite> select * from california;
Berkeley|CA|12000|1878|18.0
Los Angeles|CA|3971000|1850|503.0
San Francisco|CA|871000|1850|231.0
```

**Q2: Younger**

Create a new table `younger`, which contains the names and populations of all cities founded after 1840.

```
    create table younger as
        SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

The answer should be ordered by the population density (pop/area) of the cities.

```
sqlite3> select * from younger;
Berkeley|12000
Phoenix|1446000
Dallas|1300000
San Francisco|871000
Seattle|609000
Los Angeles|3971000
```

**Q3: Same**

Write a query that lists pairs of cities that are in the same state.

```
    create table same as
        SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

To avoid duplicate pairs, display the city with the larger area first.

```
sqlite> select * from same;
Houston|Dallas
Houston|San Antonio
Los Angeles|Berkeley
Los Angeles|San Francisco
San Antonio|Dallas
San Francisco|Berkeley
```

**Q4: Percentages**

Write a query that selects the names of every city and the city's percentage of its state population. Order the output in order of that percentage.

```
    create table percentages as
        SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";




sqlite> select * from percentages;
Berkeley|0.0305732484076433
San Francisco|2.21910828025478
Jacksonville|3.98796817387929
Dallas|4.66568567634497
San Antonio|5.1573771668521
Houston|7.53490650683703
Seattle|8.35620197585071
Los Angeles|10.1171974522293
Philadelphia|12.2397940465236
Chicago|21.0608546207328
Phoenix|21.1775043936731
New York City|43.1927254357161
```

**Q5: Num Meetings**

Here are our tables for the next two problems. We have only provided you with the headers/columns for the tables which should be sufficient:

records: Employee Name Division Title Salary Supervisor

meetings: Division Day Time

Write a query that outputs the days of the week for which fewer than 5 employees have a meeting. You may assume no department has more than one meeting on a given day.

```
SELECT "YOUR CODE HERE"
```

**Q6: Supervisor Sum Salary**

Write a query that outputs each supervisor and the sum of salaries of all the employees they supervise.

```
SELECT "YOUR CODE HERE"
```

# Submit Attendance

You're done! Excellent work this week. Please be sure to ask your section TA for the attendance form link and fill it out for credit. (one submission per person per section).

# Extra Challenge

## Pizza Time

The `pizzas` table contains the names, opening, and closing hours of great pizza places in Berkeley. The `meals` table contains typical meal times (for college students). A pizza place is open for a meal if the meal time is at or within the `open` and `close` times.

```
CREATE TABLE pizzas AS
  SELECT "Artichoke" AS name, 12 AS open, 15 AS close UNION
  SELECT "La Val's"         , 11        , 22          UNION
  SELECT "Sliver"           , 11        , 20          UNION
  SELECT "Cheeseboard"      , 16        , 23          UNION
  SELECT "Emilia's"         , 13        , 18;

CREATE TABLE meals AS
  SELECT "breakfast" AS meal, 11 AS time UNION
  SELECT "lunch"            , 13         UNION
  SELECT "dinner"           , 19         UNION
  SELECT "snack"            , 22;
```

### Q7: Open Early

You'd like to have pizza before 13 o'clock (1pm). Create a `opening` table with the names of all pizza places that `open` before 13 o'clock, listed in reverse alphabetical order.

**`opening`** table:

| name |
|---|
| Sliver |
| La Val's |
| Artichoke |

```
-- Pizza places that open before 1pm in alphabetical order



SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

### Q8: Study Session

You're planning to study at a pizza place from the moment it opens until 14 o'clock (2pm). Create a table `study` with two columns, the `name` of each pizza place and the `duration` of the study session you would have if you studied there (the difference between when it opens and 14 o'clock). For pizza places that are not open before 2pm, the `duration` should be zero. Order the rows by decreasing duration.

**Hint:** Use an expression of the form `MAX(_, 0)` to make sure a result is not below 0.

`study` table:

| name | duration |
|---|---|
| La Val's | 3 |
| Sliver | 3 |
| Artichoke | 2 |
| Emilia's | 1 |
| Cheeseboard | 0 |

```
-- Pizza places and the duration of a study break that ends at 14 o'clock



SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

### Q9: Late Night Snack

What's still open for a late night `snack`? Create a `late` table with one column named `status` that has a sentence describing the closing time of each pizza place that closes at or after `snack` time. **Important:** Don't use any numbers in your SQL query! Instead, use a join to compare each restaurant's closing time to the time of a snack. The rows may appear in any order.

`late` table:

| status |
|---|
| Cheeseboard closes at 23 |
| La Val's closes at 22 |

The `||` operator in SQL concatenates two strings together, just like `+` in Python.

```
-- Pizza places that are open for late-night-snack time and when they close



  SELECT ____ || " closes at " || ____ AS status
  FROM ____
  WHERE ____;
```

### Q10: Double Pizza

If two meals are more than 6 hours apart, then there's nothing wrong with going to the same pizza place for both, right? Create a `double` table with three columns. The `first` column is the earlier meal, the `second` column is the later meal, and the `name` column is the name of a pizza place. Only include rows that describe two meals that are **more than 6 hours apart** and a pizza place that is open for both of the meals. The rows may appear in any order.

`double` table:

| first | second | name |
|---|---|---|
| breakfast | dinner | La Val's |
| breakfast | dinner | Sliver |
| breakfast | snack | La Val's |
| lunch | snack | La Val's |

```
-- Two meals at the same place




  SELECT ____ AS first, ____ AS second, name
  FROM ____, ____, pizzas
  WHERE ____;
```