Lecture 25: SQL II

Tyler Lam
CS61A Summer 2023
2 August 2023

Announcements

- Lab 12 is due **Thursday, 08/03 (Tomorrow)**
- Homework 6 is due **Thursday, 08/03 (Tomorrow)**
- Scheme
  - Checkpoint 2 is due **Friday, 08/04**
  - Full project is due **Tuesday, 08/08**
- Scheme Contest
  - If you want to participate, the submission deadline is **Friday, 08/04**
- This is the last lecture in the course that will contribute to any assignment content!
  - After this we have three special topics lectures—these are in scope for the final, but won't introduce any new coding or problem-solving techniques. Any final exam question corresponding to the special topics should be answerable so long as you watched the lecture.

About Me!

- I’m a third-year majoring in Computer Science and Data Science, Minor in Linguistics
- 3x Tutor, 1st time TA
- I like:
    - Traveling
    - Trees
    - Trains

Review

SQL

SQL (**S**tructured **Q**uery **L**anguage) is a declarative programming language we can use to "query" information from databases.

SQL assumes data is organized in rows and columns. Each row corresponds to a unique data point, and each column corresponds to a features of each data point.

We could have a database on any kind of information that we want!

Databases

In this section of the lecture, we'll primarily work with a database containing two tables:

`cities`, which contains four columns:

- `city`
- `county`
- `state`
- `pop_2020`

`pop_area_2010`, which contains five columns:

- `city`
- `county`
- `state`
- `pop_2010`
- `land_area_sq_miles`

These tables will be made public in a Jupyter Notebook after lecture if you want to refer to it later on.

Querying a Database

```
sql> SELECT * FROM cities LIMIT 3;
city|county|state|pop_2020
Antioch|Contra Costa|CA|115291
Berkeley|Alameda|CA|124321
Berkeley|Cook|IL|5338
sql> SELECT * FROM pop_area_2010 LIMIT 3;
Antioch|Contra Costa|CA|102372|29.17
Berkeley|Alameda|CA|112580|10.43
Berkeley|Cook|IL|5209|1.4
sql> SELECT * FROM pop_area_2010 ORDER BY pop_2010 DESC LIMIT 3;
Los Angeles|Los Angeles|CA|3792621|469.49
San Jose|Santa Clara|CA|945942|178.24
San Francisco|San Francisco|CA|805235|46.9
sql> SELECT * FROM pop_area_2020 WHERE pop_2010 < 5000;
city|county|state|pop_2010|land_area_sq_miles
Springfield|Baca|CO|1451|1.13
Springfield|Effingham|GA|2852|3.25
```

Joins

```
sql> SELECT a.city, pop_2010, pop_2020 FROM cities AS a,
pop_area_2010 AS b WHERE a.city = b.city AND a.county = b.county AND
a.state = b.state;
city|pop_2010|pop_2020
Antioch|102372|115291
Berkeley|112580|124321
Berkeley|5209|5338
Berkeley|41255|43754
...
Palatine|68557|67908
sql> SELECT a.city, b.city FROM cities AS a, cities AS b WHERE
a.county = b.county AND a.city < b.city;
city|city
Antioch|Concord
Antioch|Richmond
Berkeley|Fremont
Berkeley|Hayward
Berkeley|Oakland
```

Numerical and String Expressions

Expressions

We don't have to SELECT information directly from columns in SQL—we can also use operations to transform that data.

For example, we can do string concatenation on information in our database, using the || operator.

```
SELECT (city || ", " || state) AS full_city, (county
|| " County") AS full_county FROM cities LIMIT 5;
```

full_city	full_county
Antioch, CA	Contra Costa County
Berkeley, CA	Alameda County
Berkeley, IL	Cook County
Berkeley, NJ	Ocean County
Berkeley, MO	St. Louis County

Expressions

We can also use function calls and arithmetic operators to create **numeric expressions** that do operations on our data.

- Combining values: +, -, *, /, %, and, or,
- Transforming values: ABS( ), ROUND( ), NOT, -
- <, <=, >, >=, <>, !=, =

```
SELECT pop_2020 / 1000 AS pop_1000s FROM cities;
```

SQL also has the ability to do some operations on strings!

- Concatenating strings: || (we just saw this one)
- Selecting substrings: SUBSTR( )

```
SELECT
   SUBSTR(city, 1, 3) AS initials FROM cities;
```

Exercise!

Let's find the city that has gained the most people between 2010 and 2020.

Select a table with two columns:

- Every `city` in the `cities` table and the `pop_area_2010` table
- The difference between `pop_2020` and `pop_2010` of that city
- Sort in descending order by that difference

My solution:

```
SELECT
    a.city, a.pop_2020 - b.pop_2010 AS pop_difference
    FROM cities AS a, pop_area_2010 AS b
    WHERE a.city = b.city AND a.county = b.county AND a.state
    = b.state ORDER BY pop_difference DESC LIMIT 1;
```

Aggregation

Aggregate Functions

So far, all the functions we've been able to write have only operated on single rows at a time.

If we wanted to compare multiple rows across a table, we could use joins, but even then we could only really compare as many rows as joins we were willing to do.

However, SQL has **aggregation functions** that allow us to compare across all the rows in a table

```
SELECT pop_2020 FROM cities ORDER BY pop_2020 DESC LIMIT 1;
```

pop_2020
3898747

```
SELECT MAX(pop_2020) FROM cities;
```

max(pop_2020)
3898747

Mixing Aggregation and Single Values

If we include a non-aggregated column in an aggregation query, SQL will still fill that column with a value

In the case of MAX and MIN, that column will be filled based on the row in which the maximum or minimum value lives

For other aggregation functions (SUM, AVG, etc.) a value will be picked arbitrarily

```
SELECT city_name, MAX(pop_2010) FROM pop_area_2010;

SELECT city_name, MIN(pop_2010) FROM pop_area_2010;

SELECT city_name, AVG(pop_2010) FROM pop_area_2010;
```

city	max(pop_2010)
Los Angeles	3792621

city	min(pop_2010)
Springfield	1451

city	avg(pop_2010)
Antioch	304744.5

Aggregating Over Expressions

You can also aggregate over expressions:

```
SELECT a.city, b.city, MAX(a.pop_2010 - b.pop_2010)
   FROM pop_2010 AS a, pop_2010 AS b;
```

What do we think this query does?

a.city	b.city	max(a.pop_2010 - b.pop_2010)
Los Angeles	Springfield	3791170

Break

Grouping

Taylor Swift Discography

Taylor Swift
Taylor Swift
FEARLESS
Speak Now
Taylor Swift
RED
TAYLOR SWIFT
1989
Taylor Swift
D.L.X.
reputation
Lover
Midnights

Creating Tables From Tables

What does this SQL statement do?

```
CREATE TABLE swiftsongs AS
   SELECT name, album, (length / 1000) AS seconds
   FROM taylorswift;
```

name	album	seconds
...Ready For It?	reputation	208
22	Red (Deluxe Edition)	232
A Perfectly Good Heart	Taylor Swift	220
A Place in this World	Taylor Swift	199
Afterglow	Lover	223
All Too Well	Red (Deluxe Edition)	329

...(178 more rows)

Grouping

We can divide our table into **groups**, and then aggregate within those groups, instead of aggregating across our entire table.

We do this using the GROUP BY clause:

```
SELECT name, MAX(seconds) FROM swiftsongs
   GROUP BY album;
```

name	album	seconds
Clean	1989 (Deluxe)	271
Bad Blood	1989 (Deluxe)	211
Untouchable (Taylor's Version)	Fearless (Taylor's Version)	312
Daylight	Lover	293
The Man	Lover	190

name	MAX(seconds)
Clean	271
Untouchable (Taylor's Version)	312
Daylight	293
Snow On The Beach (feat. Lana Del Rey)	256
All Too Well	329

Aggregation Functions and Grouping

These are the main aggregation functions you need to know for this class (and life in general)

- `MAX(<col>)` - Finds the maximum value of `<col>`, within a group
- `MIN(<col>)` - Finds the minimum value of `<col>`, within a group
- `SUM(<col>)` - Adds together all the values in `<col>`, within a group
- `AVG(<col>)` - Finds the average of all the values in `<col>`, within a group
- `COUNT(*)` - Counts the number of elements in a group

name	MAX(seconds)
Dear John	403

name	MIN(seconds)
I Wish You Would - Voice Memo	107

name	SUM(seconds)
…Ready For It?	43038

name	AVG(seconds)
…Ready For It?	233.902173913043

name	COUNT(*)
...Ready For It?	184

Exercise!

Create a table with two columns:

- The name of each distinct album in the helper table we made named `swiftsongs`
- The number of songs each distinct album has

My solution:

```
SELECT album, COUNT(*) FROM swiftsongs GROUP BY album;
```

album	**COUNT(*)**
1989 (Deluxe)	19
Fearless (Taylor's Version)	26
Lover	18
Midnights	13

Grouping by Multiple Columns

We don't have to just group by single columns—we can also group by multiple columns!

```
    SELECT album, instrumentalness, COUNT(*) FROM
taylorswift GROUP BY album, instrumentalness;
```

album	instrumentalness	COUNT(*)
1989 (Deluxe)	0	8
1989 (Deluxe)	1.64e-06	1
1989 (Deluxe)	6.16e-06	1
Fearless (Taylor's Version)	0	24
Fearless (Taylor's Version)	3.97e-06	1
Fearless (Taylor's Version)	1.2e-05	1

Grouping by expressions

We can also group by expressions!

```
SELECT
    SUBSTR(name, 1, 1) AS first_character, seconds / 60 AS
    minutes FROM swiftsongs GROUP BY
    SUBSTR(name, 1, 1), seconds / 60;
```

first_character	minutes
.	3
2	3
A	3
A	5
B	2

Filtering groups

We can also filter groups based on criteria using the HAVING clause

```
SELECT album, COUNT(*) FROM swiftsongs GROUP BY album
    HAVING COUNT(*) > 20;
```

album	COUNT(*)
Fearless (Taylor's Version)	26
Red (Deluxe Edition)	22
Speak Now (Deluxe Package)	22

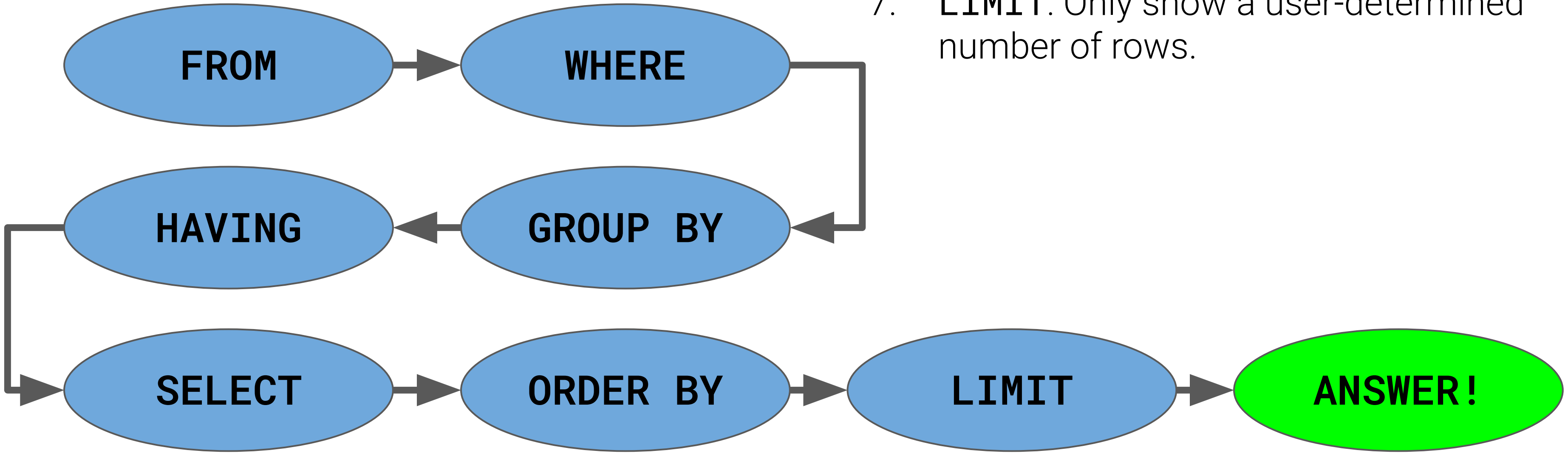
HAVING is similar to WHERE, but specifically for filtering by aggregate functions. If you're using an aggregate function in your filter clause, you should use HAVING. Otherwise, you should use WHERE

You can also use both together!

Final Query Structure and Order of Operations

```
SELECT <column/expression list>
FROM <table>
[WHERE <condition>]
[GROUP BY <column(s)>]
[HAVING <condition>]
[ORDER BY <column(s)> [DESC/ASC]]
[LIMIT <number of rows>];
```

1. `FROM`: Retrieve the tables.
2. `WHERE`: Filter the rows.
3. `GROUP BY`: Make groups.
4. `HAVING`: Filter the groups.
5. `SELECT`: Aggregate into rows and get specific columns.
6. `ORDER BY`: Sort by certain columns (optionally ascending/descending, default is ascending).
7. `LIMIT`: Only show a user-determined number of rows.



Final Exercise

Which album has the highest average danceability of all albums that contain over 15 songs and are at least 1 hour long?

```
SELECT album, COUNT(*), SUM(length / 1000 / 60) AS minutes,
AVG(danceability) AS danceability FROM taylorswift GROUP BY
album HAVING COUNT(*) > 15 AND SUM(minutes) >= 60 ORDER BY
danceability DESC LIMIT 1;
```

album	COUNT(*)	minutes	danceability
Red (Deluxe Edition)	22	79	0.633409090909091

HAVING is similar to WHERE, but specifically for filtering by aggregate functions. If you're using an aggregate function in your filter clause, you should use HAVING. Otherwise, you should use WHERE

You can also use both together!

Final Exercise

Which album has the highest average danceability of all albums that contain over 15 songs and are at least 1 hour long?

```
SELECT album, COUNT(*), SUM(length / 1000 / 60) AS minutes,
AVG(danceability) AS danceability FROM taylorswift GROUP BY
album HAVING COUNT(*) > 15 AND SUM(minutes) >= 60 ORDER BY
danceability DESC LIMIT 1;
```

album	COUNT(*)	minutes	danceability
Red (Deluxe Edition)	22	79	0.633409090909091

HAVING is similar to WHERE, but specifically for filtering by aggregate functions. If you're using an aggregate function in your filter clause, you should use HAVING. Otherwise, you should use WHERE

You can also use both together!

Database Connections

Creating, dropping, and modifying tables

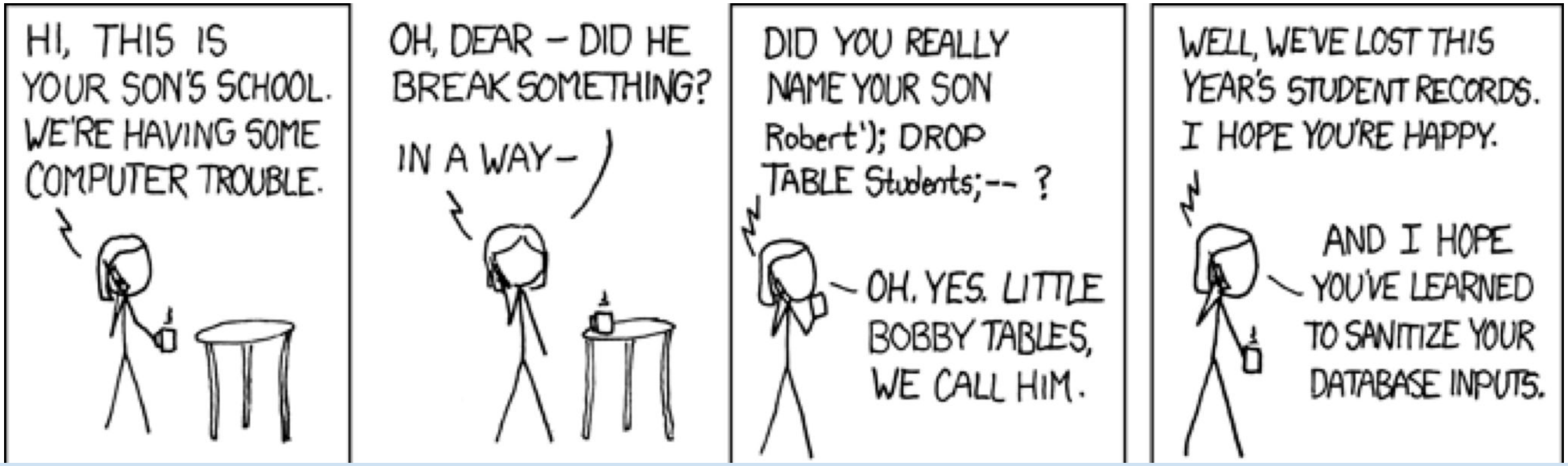
```
CREATE TABLE [table]([column-defs]);

DROP TABLE [table];

INSERT INTO [table] VALUES ([exprs]);

UPDATE [table] SET [column-city] = [expr] WHERE
[expr];

DELETE FROM [table] WHERE [expr];
```

None of these functions are in-scope for assignments or exams in this class, but the ability to modify tables is crucial if you're ever dealing with SQL in your future endeavors!

Python and SQL

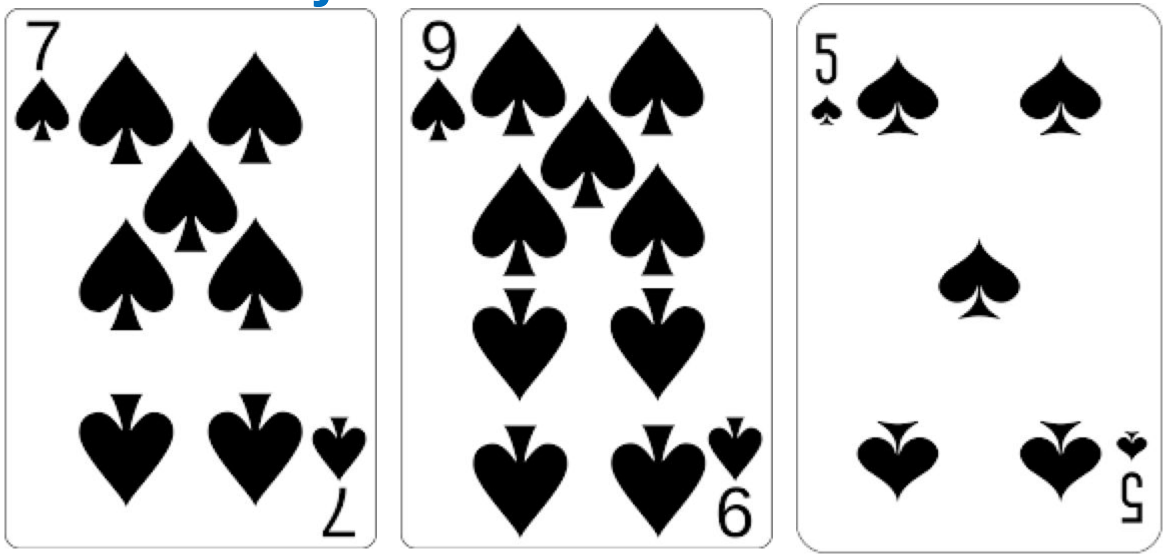
Python has a module called `sqlite3` that lets us interact with SQL databases! (truly modules for everything)

```
db = sqlite3.Connection('cards.db')
sql = db.execute
sql('DROP TABLE IF EXISTS cards;')
sql('CREATE TABLE cards(card, place);')
def play(card, place):
    sql('INSERT INTO cards VALUES (?, ?)', (card, place))
    db.commit()
```

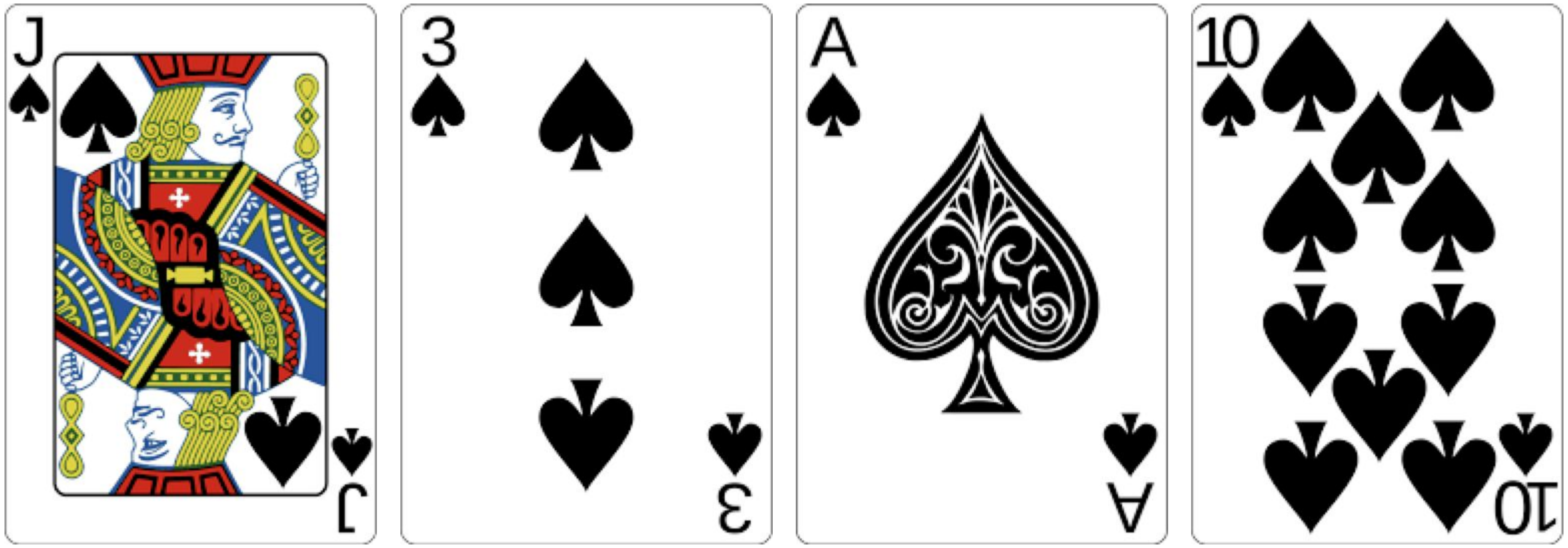

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Casino Blackjack

Player



Dealer



Connections to Other Fields

SQL is an amazing tool for managing databases and is used extensively throughout the Computing and Data Science world.

If you liked the SQL portion of this course, consider taking:

- DATA 8, DATA 100, DATA 101
- COMPSCI 186

Feel free to talk with me about the Data Science Major!