CS 61A Discussion 11

Structured Query Language

April 20, 2017

^lol

select a* from content;

ANNOUNCEMENTS

- Scheme project is due today
  - Do read your autograder emails!
  - No composition revisions! :(
- Ants composition revisions are due on 4/30
- Scheme recursive art!!! (due 5/1)
- Databricks lab next Tuesday

ATTENDANCE

What shall the Christians do today?

- *`tiny.cc/420praiseit`*

AGENDA

- S
- Q
- L

ADVICE

To quit the sqlite3 interpreter, run `.quit` (mostly a note from last semester for when I forgot it again this semester)

select sql_intro from content;

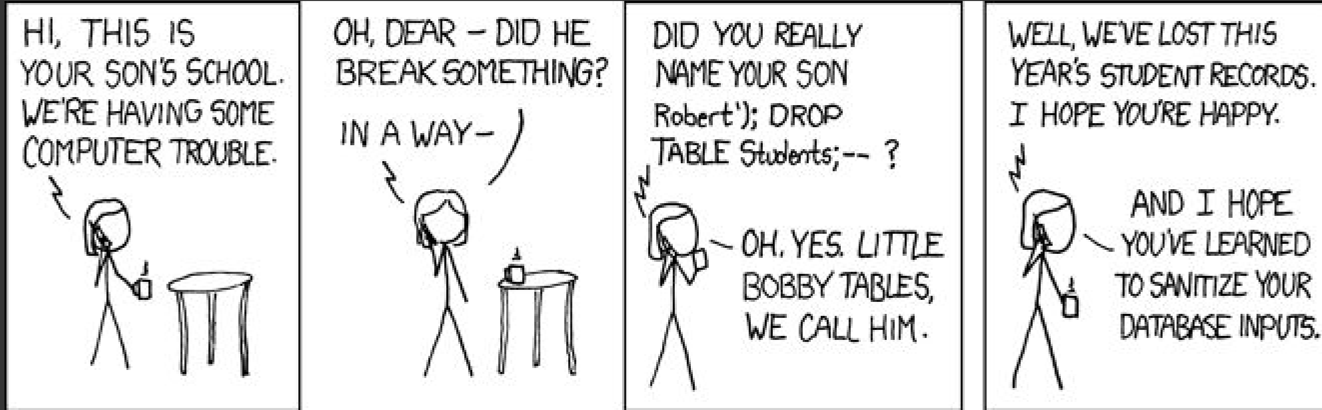
SQL is a declarative language for managing database systems.

- This mostly revolves around **creating tables** and **making queries** into them.
- "*Declarative*" - I tell you **what** I want. You do it for me. I don't care **how**.

Here's what some ~~previous 61A students~~ satisfied customers have to say about it:

- "*My mom uses SQL*" - anonymous Fall 2016 student
- "*I don't remember SQL at all*" - anonymous Spring 2016 student
- "*I love SQL! I SQL all day every day and I'm glad I studied it in 61A. Also Owen is the best*" - anonymous and not at all fake person
- "*The sequel to what?*" - anonymous Fall 2014 student

select xkcd:) from content;



`select frequently_asked from content;`

- Why the obscenity do we need SQL when we can just write for-loops and achieve the same result?
  - SQL is **uniquely** and **especially** optimized for information storage and retrieval. To learn more about this, take CS 186.
    - Would you rather eat a meal prepared by a professional chef (SQL) or by your lazy friend (whose instrument of choice is a microwave – for-loops)?
    - ...would you rather eat a bagel from a specialty bagel shop (SQL) or from Target (for-loops)?
- Does anyone actually use SQL?
  - Yes
    - Fall 2016 student’s mom
    - Facebook for its user data (I have not fact-checked this)
    - OK for all of its data

select the_basics from content;

In SQL, data is organized into tables.

- **table**: a bunch of data in a single structure
- **column**: all of the values for a specific data attribute
- **row**: a “table entry” (with a value for every column)

TABLE Football

Berkeley	Stanford	Year
30	7	2002
28	16	2003
17	38	2014

`select select from content;`

Want to make a query? Your buddy `select` can help you out.

```
SELECT <column expression(s)>
  FROM <table(s)>
[WHERE <predicate(s)>]
[GROUP BY <column expression(s)>
  [HAVING <predicate(s)>]]
[ORDER BY <column expression(s)>]
[LIMIT <limit>];
```

`[]` means “optional”, `<>` means “insert actual content”

select an_explanation from content;

Evaluation (mostly) happens in the order in which it is written.

```
SELECT <column expression(s)>     “we’ll want these columns as output”
  FROM <table(s)>     “from these tables”
[WHERE <predicate(s)>]     “but only values that satisfy these conditions”
[GROUP BY <column expression(s)>     “and also only one value per group”
  [HAVING <predicate(s)>]]     “or actually per group that satisfies these conditions”
[ORDER BY <column expression(s)>]     “...hmm. order the output like so”
[LIMIT <limit>];     “and finally limit our output to some number of entries”
```

`select assorted_notes from content;`

- Using the asterisk (*) as the columns will select ALL of them
- Whitespace and capitalization of keywords is unimportant
- `where` filters **rows**. `having` filters **groups**. More on this in a sec...

heheheh

select sql_groups0 from content;

```
[GROUP BY <column expression(s)>
  [HAVING <predicate(s)>]]
```

Grouping: used for aggregation. When we say `GROUP BY X`, every row with the same value of `X` will be put into one group. Accordingly, there will be a group for every distinct value of `X`. Note that only **one value per group** can contribute to the output.

Default group: everything

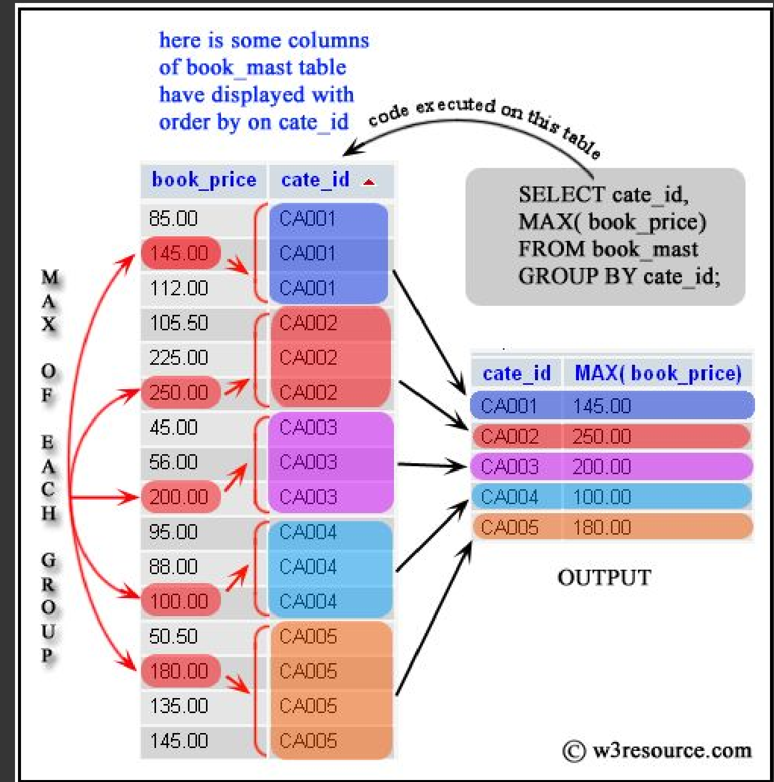
Aggregate functions will be applied within individual groups:

`count, max, min, sum, avg, first, last` ← vaguely ordered in terms of 61A importance

select sql_groups1 from content;

HAVING filters out groups (by contrast, WHERE filters out individual rows)

tl;dr Grouping is like dividing your data into buckets and then only using one aggregated row per bucket



select sql_ordering from content;

...ORDER BY <column expression(s)>...

***To output in descending order**, you can use*

ORDER BY <column expression(s)> DESC

or

ORDER BY -<column expression(s)>

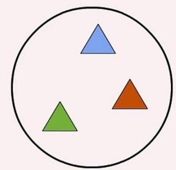
if the column expression is numerical

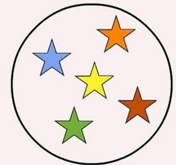
`select sql_joins from content;`

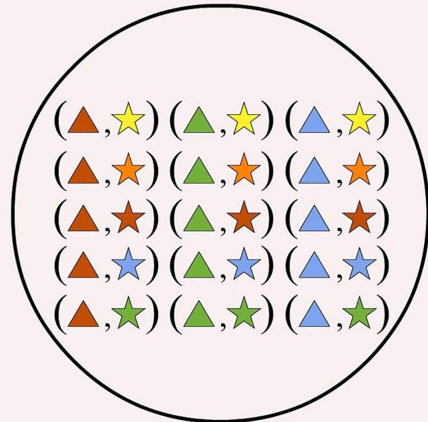
- You can think of a join as being the Cartesian product of the table rows (each row from each table combined with each row from every other table).

- i.e. the result of a join is a “super”-table, *where every row from the first table is paired with every row from the second table*!

- Aliasing (`<table> as <name>`) never really hurts. Unless you have arthritic fingers and typing extra characters hurts. :(If there are any similarly-named columns across your tables, you can *just do it*.







select recursive_queries0 from content;

+ Create a local table using `with`
+ Add base cases to the table (starter rows, e.g. a row with 0 and 1 if we're talking Fibonacci numbers)
+ Reference the table recursively using `SELECT` statements; have some kind of stopping point for this recursion as a `WHERE` condition

```
create table naturals_leq5 as
    with num(n) as (
        SELECT 0 UNION
        SELECT n + 1 FROM num WHERE n < 5
    )
    SELECT * from num;
```

`select local_tables from content;`

```
with [local-tables] select [columns] from [tables]
    where [condition] order by [criteria]
```

Local tables only exist for the sake of the main `select` statement; think of them as “helper tables” that just so happen to support recursive construction (which is generally what we use them for).

select recursive_queries1 from content;

Fibonacci example:

```
with fibonacci(prev, curr) as (
    select 0, 1 union
    select curr, prev + curr from fibonacci where curr < 200
) select prev from fibonacci;
```

We *need* a stopping point for our recursion!
(hence the < 200)

Recursive SQL - recursive queries

```
sqlite> create table naturals as
   ...>   with num(n) as (
   ...>     select 0 union
   ...>     select n + 1 from num where n < 5
   ...>   )
   ...>   select * from num;
```

The basic algorithm for computing the content of the recursive table is as follows:

1. **Run the initial-select** and **add the results to a queue**.
2. While the **queue is not empty**:
    a. Extract a **single row from the queue**.
    b. Insert that **single row into the recursive table**
    c. Pretend that the single row just extracted is the only row in the recursive table and run the recursive-select, adding all results to the queue.

I stole this slide from Eric Pai (source)