CS 61A Discussion 10

Structured Query Language

November 17, 2016

`select announcements, agenda from content;`

ANNOUNCEMENTS

+ Scheme project is due today
+ It's almost the end of the semester
+ It's almost 2017
+ It's almost 2018
+ It's almost 2094
+ Well that's kind of sobering
+ : (

AGENDA

+ S,
+ Q,
+ and L

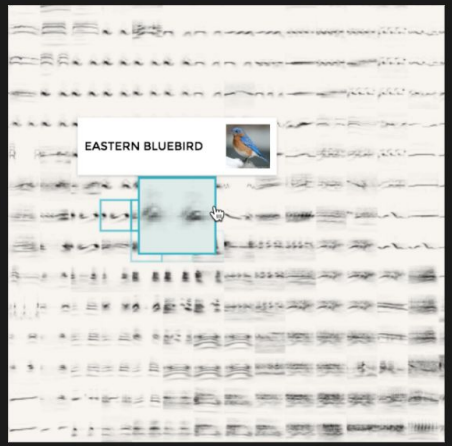
ADVICE

+ To quit the sqlite3 interpreter, run `.quit` (mostly a note for when I forget this again next semester)

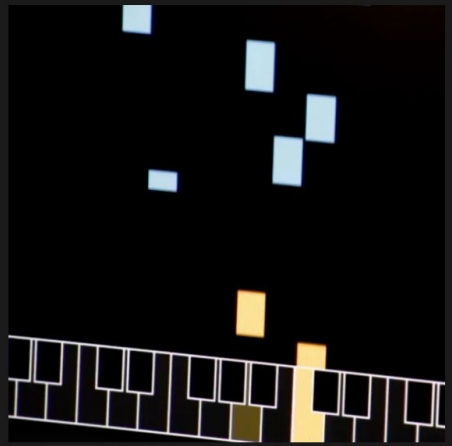
select ai_experiments from content;

Check this out if you haven't seen it (who isn't interested in AI these days?):
https://aiexperiments.withgoogle.com/

Thing Translator



Bird Sounds



A.I. Duet

`select sql_intro from content;`

SQL is a declarative programming language for managing database systems.
"*Declarative*" - I tell you what I want. You get (or do) it for me. I don't care how.

Past and current CS 61A students on SQL:

+ "*SQL is fine*" - anonymous Fall 2016 student
+ "*I didn't even really work on the lab*" - anonymous Fall 2016 student
+ "*I don't remember SQL at all*" - recent 61A graduate
+ "*The sequel to what?*" - not-so-recent 61A graduate

select basic_terminology from content;

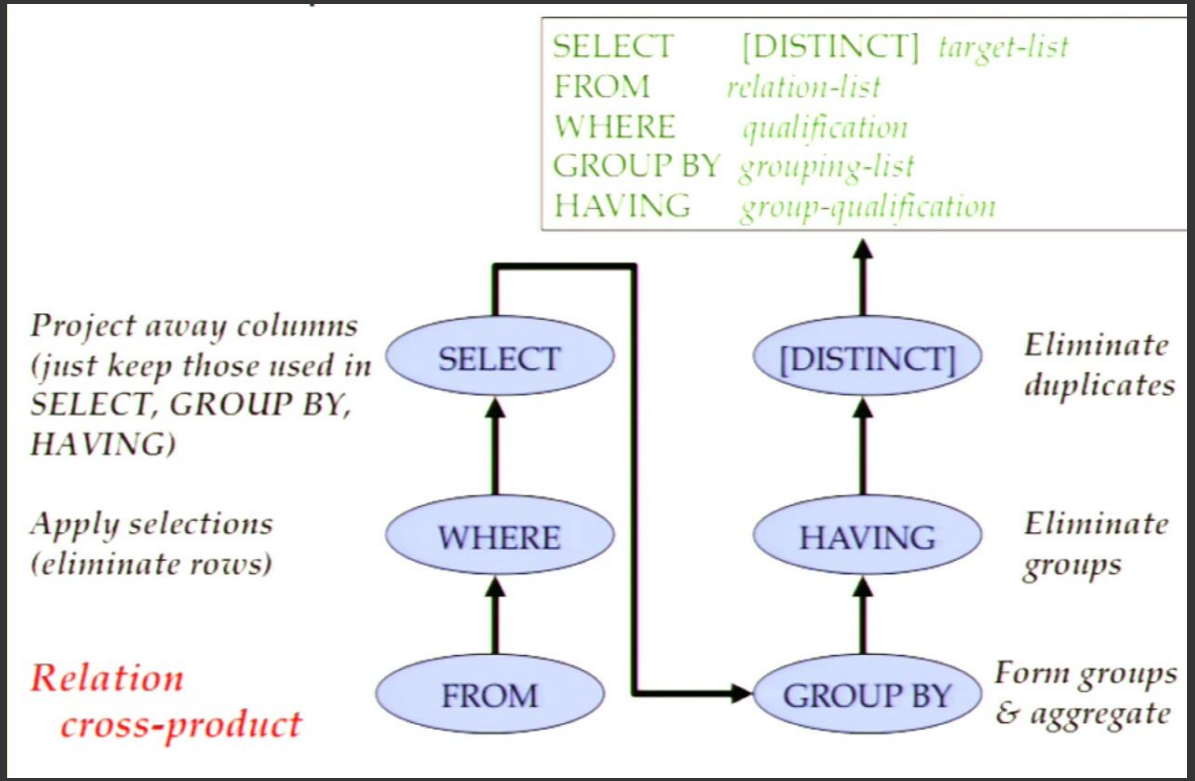
- Table: a bunch of data in a single structure
- Column: a **category** or **type** that we can have data values for (technically a column would be all of the values for one type)
- Row: a single data entry in a table (contains a value for every column)

TABLE Football

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

`select sql_queries from content;`

CS 186 visualization of the `SELECT` evaluation pipeline →



select sql_queries2 from content;

```
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>];
```

[]: optional
<>: insert actual content

`select sql_queries3 from content;`

Evaluation pretty much happens in the order it's written.

```
SELECT <column expression(s)>   "we'll want this stuff as output"
  FROM <table(s)>   "from these tables"
[WHERE <predicate(s)>]   "but only the stuff that satisfies these conditions"
[GROUP BY <column expression(s)>   "and also only one value per group"
  [HAVING <predicate(s)>]]   "actually per group that satisfies these conditions"
[ORDER BY <column expression(s)>]   "...order the output like so"
[LIMIT <limit>];   "then finally limit it to some number of entries"
```

`select sql_groups 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

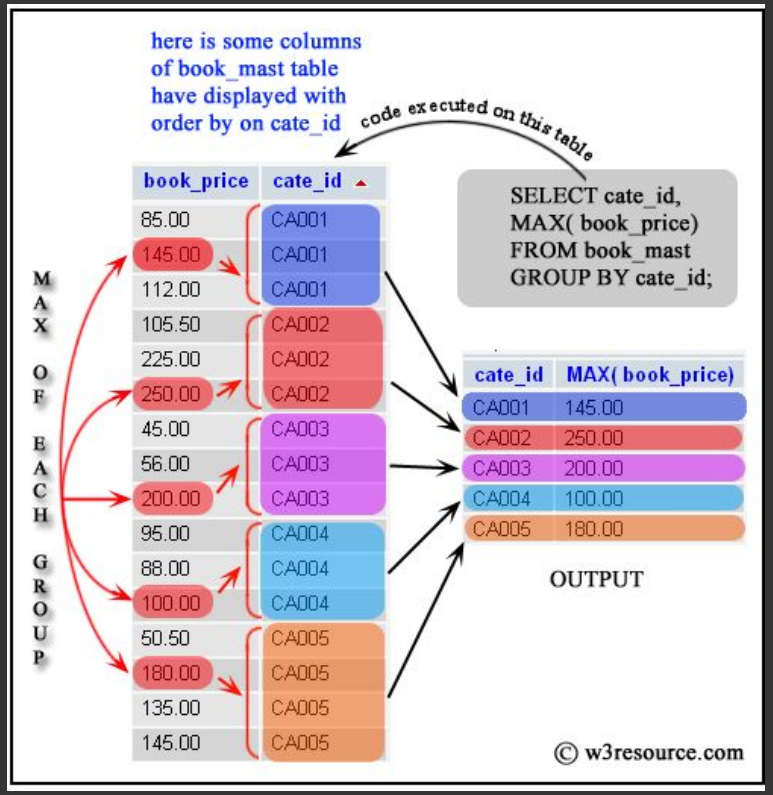
Aggregate functions will be applied within individual groups:

`count, max, min, sum, avg, first, last` ← vague order of 61A importance

`select sql_groups2 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)>...

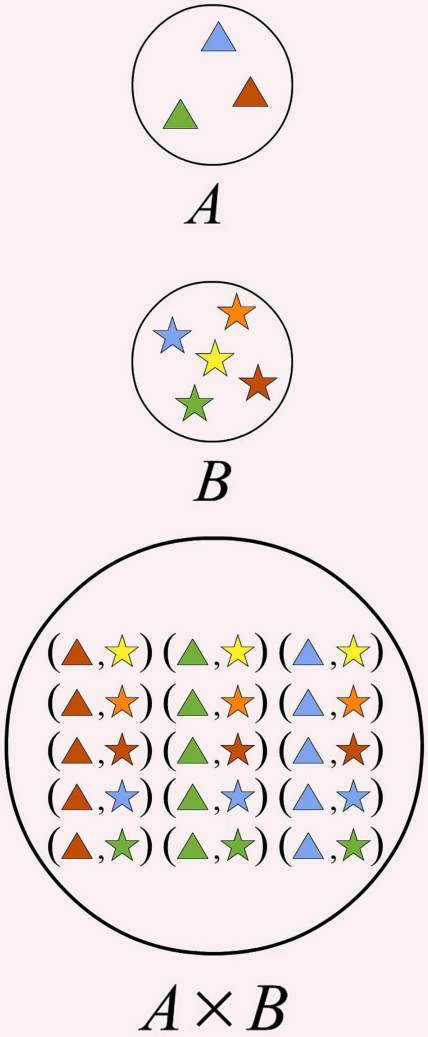
To output in descending order, you can use
ORDER BY <column expression(s)> DESC
or
ORDER BY -<column expression(s)>
if the column expression is numerical

`select sql_joins from content;`

+ The only join you need to know is the **cross join**.
+ In 61A we call it the **join**, period.

+ Thus 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).

+ Aliasing (`<table> as <name>`) never really hurts. If there are any similarly-named columns across your tables, you can just do it.



`select recursive_queries 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 recursive_queries2 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)