Computational Structures in Data Science

SQL: Aggregrations

UC Berkeley

cc by nc ©

Announcements

- Extensions:
  - Final Date **all extensions**: End of Dead Week (5/8)
  - Please try not to take that long. ☺
- Ants Oral Check Offs Thos Week (and next)!
  - Reminder to sign up for a check off time slot.
  - *Sign up early, so you get a preferred time.*
  - Go to OH if you'd like to practice!
  - You should have a submission by the time you do your check off.

© CC BY-NC-SA

Review: SQL Basics

- SQL Keywords are case-*insensitive*
  - e.g. `SELECT` and `select` do the same thing
  - I *try* to capitalize them to make it clear what's-what.
- The order of SQL keywords matters
  - e.g. `SELECT ... FROM … WHERE …`
- Every statement ends in a `;`
- Whitespace doesn't matter
  - But indentations and newlines help make queries readable!
- Despite being a standard, differences do exist between databases.

We use `sqlite3`.

 © CC BY-NC-SA

Computational Structures in Data Science

Order Of Execution Revisited

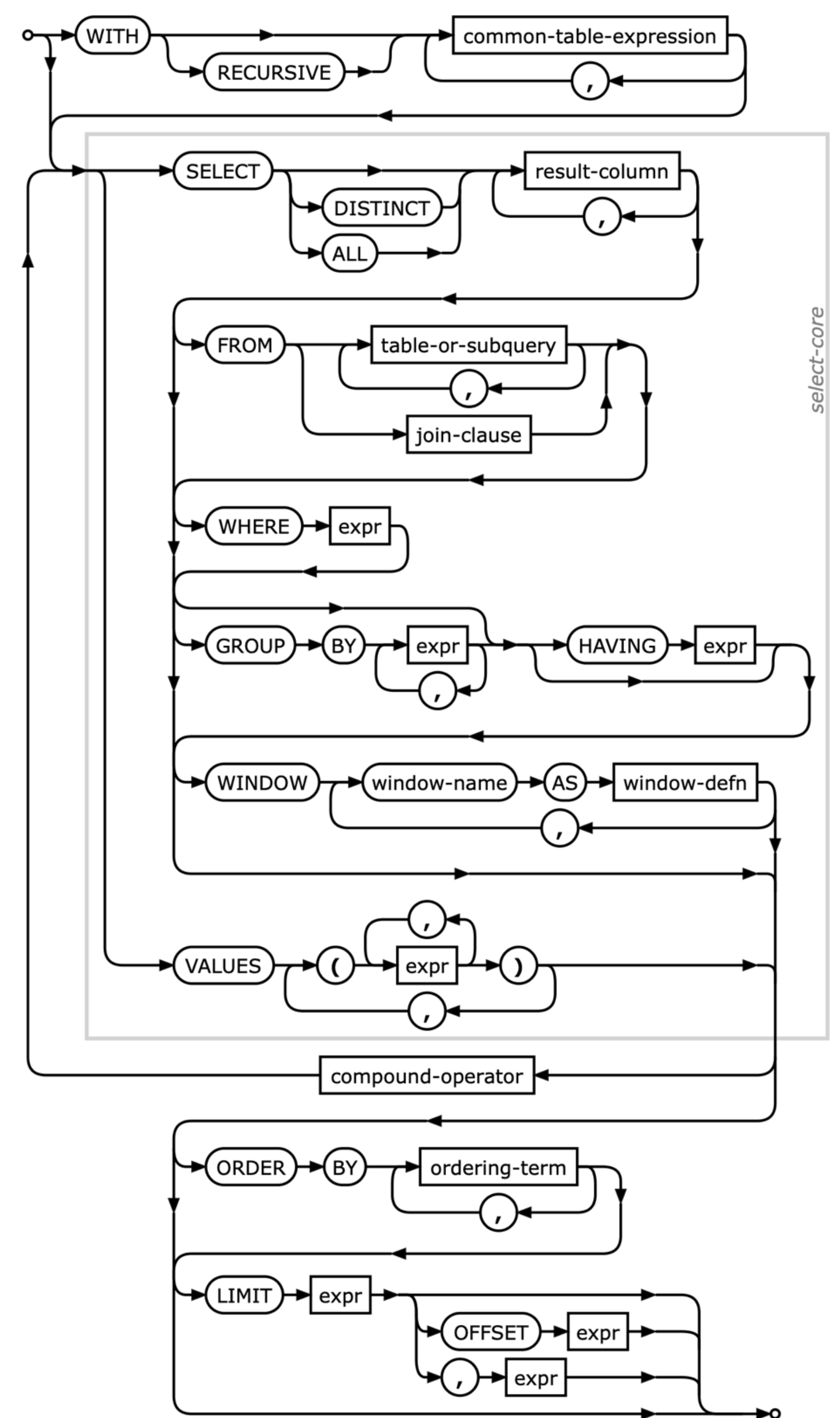
UC Berkeley

Michael Ball | UC Berkeley | https://c88c.org | © CC BY-NC-SA

Anatomy of a SQL Statement

From the SQLite Docs

(Don't memorize, but a fairly easy to read reference)



SQL Order Of Execution (for now)

```
• SELECT S            3
• FROM R1, R2, …      1
• WHERE C1            2
```

A **SELECT FROM WHERE (SFW)** query is the most standard SQL query structure that extract records from a relation.

Order of execution:

1. **FROM**: Fetch the tables and compute the cross product of R1, R2, ...
2. **WHERE**: “Row filter.” For each tuple from step 1, keep only those that satisfy condition C1
3. **SELECT**: add to output based on S

SQL Order Of Execution (for now)

- SELECT S — 3
- FROM R1, R2, … — 1
- WHERE C1 — 2
- ORDER BY O1 — 4
- LIMIT X — 5

A **SELECT FROM WHERE (SFW)** query is the most standard SQL query structure that extract records from a relation.

Order of execution:

1. **FROM**: Fetch the tables and compute the cross product of R1, R2, ...
2. **WHERE**: "Row filter." For each tuple from step 1, keep only those that satisfy condition C1
3. **SELECT**: add to output based on S
4. **ORDER BY**: Sort all the rows according to the conditions O1, etc.
5. **LIMIT**: The final result to the desired number of rows

SQL "Order of Execution," approximately

- `SELECT S` 5
- `FROM R1, R2, …` 1
- `WHERE C1` 2
- `GROUP BY A1, A2, …` 3
- `HAVING C2` 4

Order of execution:

- **FROM**: Fetch the tables and compute the cross product of R1, R2, ...
- **WHERE**: "Row filter." For each tuple from step 1,
keep only those that satisfy condition C1
- **GROUP BY**: A1, A2, ...
- For each group, compute all aggregates needed in C2 and S
- **HAVING**: For each group, check if C2 is satisfied
- **SELECT**: add to output based on S

Computational Structures in Data Science

SQL: Joins

UC Berkeley

cc by nc ©

Joining tables

- Two tables are joined by a comma to yield all combinations of a row from each
  - select * from sales, cones;

```
create table sales as
    select "Baskin" as Cashier, 1 as TID union
    select "Baskin", 3 union
    select "Baskin", 4 union
    select "Robin", 2 union
    select "Robin", 5 union
    select "Robin", 6;
```

Cashier	TID
Baskin	1
Robin	2
Baskin	3
Baskin	4
Robin	5
Robin	6

```
sales.join('TID', cones, 'ID')
```

TID	Cashier	Flavor	Color	Price
1	Baskin	strawberry	pink	3.55
2	Robin	chocolate	light brown	4.75
3	Baskin	chocolate	dark brown	5.25
4	Baskin	strawberry	pink	5.25
5	Robin	bubblegum	pink	4.75
6	Robin	chocolate	dark brown	5.25

```
sqlite> select * from sales, cones;
Baskin|1|1|strawberry|pink|3.55
Baskin|1|2|chocolate|light brown|4.75
Baskin|1|3|chocolate|dark brown|5.25
Baskin|1|4|strawberry|pink|5.25
Baskin|1|5|bubblegum|pink|4.75
Baskin|1|6|chocolate|dark brown|5.25
Baskin|3|1|strawberry|pink|3.55
Baskin|3|2|chocolate|light brown|4.75
Baskin|3|3|chocolate|dark brown|5.25
Baskin|3|4|strawberry|pink|5.25
Baskin|3|5|bubblegum|pink|4.75
Baskin|3|6|chocolate|dark brown|5.25
Baskin|4|1|strawberry|pink|3.55
Baskin|4|2|chocolate|light brown|4.75
Baskin|4|3|chocolate|dark brown|5.25
Baskin|4|4|strawberry|pink|5.25
Baskin|4|5|bubblegum|pink|4.75
Baskin|4|6|chocolate|dark brown|5.25
Robin|2|1|strawberry|pink|3.55
Robin|2|2|chocolate|light brown|4.75
Robin|2|3|chocolate|dark brown|5.25
Robin|2|4|strawberry|pink|5.25
Robin|2|5|bubblegum|pink|4.75
Robin|2|6|chocolate|dark brown|5.25
Robin|5|1|strawberry|pink|3.55
Robin|5|2|chocolate|light brown|4.75
Robin|5|3|chocolate|dark brown|5.25
Robin|5|4|strawberry|pink|5.25
Robin|5|5|bubblegum|pink|4.75
Robin|5|6|chocolate|dark brown|5.25
Robin|6|1|strawberry|pink|3.55
Robin|6|2|chocolate|light brown|4.75
Robin|6|3|chocolate|dark brown|5.25
Robin|6|4|strawberry|pink|5.25
Robin|6|5|bubblegum|pink|4.75
Robin|6|6|chocolate|dark brown|5.25
```

Joins

- Joins combine two tables
- A "cross product" or full join gives *all combiniations*
- **This is often not useful!**
- So, we can do an *inner join* where we "combine" rows only on some logical identifier, like an "id"
  - Often this is called a "foreign key" or a reference to an object in another table.

Inner Join

```
SELECT * FROM sales, cones WHERE cone_id = cones.id;
```

When column names conflict we write: `table_name.column_name` in a query.

```
sqlite> SELECT * FROM cones, sales WHERE cone_id=cones.id;
Id|Flavor|Color|Price|Cashier|id|cone_id
1|strawberry|pink|3.55|Baskin|3|1
1|strawberry|pink|3.55|Robin|6|1
2|chocolate|light brown|4.75|Baskin|1|2
2|chocolate|light brown|4.75|Baskin|4|2
2|chocolate|light brown|4.75|Robin|5|2
3|chocolate|dark brown|5.25|Robin|2|3
```

 © CC BY-NC-SA

Computational Structures in Data Science

SQL: Aggregations

UC Berkeley

CC BY-NC-SA

Unique & DISTINCT values

```
SLECT DISTINCT [columns] FROM [table] WHERE [condition];
```

```
sqlite> select distinct Flavor, Color from cones;
strawberry|pink
chocolate|light brown
chocolate|dark brown
bubblegum|pink
sqlite>
```

```
In [8]: cones.groups(['Flavor', 'Color']).drop('count')
Out[8]:
```

Flavor	Color
bubblegum	pink
chocolate	dark brown
chocolate	light brown
strawberry	pink

```
In [7]: np.unique(cones['Flavor'])
Out[7]: array(['bubblegum', 'chocolate', 'strawberry'], dtype='<U10')
```

 © CC BY-NC-SA

Grouping and Aggregations

- The `GROUP BY` clause is used to group rows returned by SELECT statement into a set of summary rows or groups based on values of columns or expressions.
- Apply an aggregate function, such as SUM, AVG, MIN, MAX or COUNT, to each group to output the summary information.

```
cones.group('Flavor')
```

Flavor	count
bubblegum	1
chocolate	3
strawberry	2

```
sqlite> select count(Price), Flavor from cones group by Flavor;
count(Price)|Flavor
1|bubblegum
2|chocolate
2|strawberry
```

```
cones.select(['Flavor', 'Price']).group('Flavor', np.mean)
```

Flavor	Price mean
bubblegum	4.75
chocolate	5.08333
strawberry	4.4

```
sqlite> select avg(Price), Flavor from cones group by Flavor;
avg(Price)|Flavor
4.75|bubblegum
5.0|chocolate
4.4|strawberry
```

Aggregate Functions

- `COUNT(*), SUM(column), MIN(...), MAX(…)`
- All these functions operate on a *group*
- By default, this group is all rows in the projection.
- E.g. COUNT(*) is the total number of rows?
  - Why do we need *? SQL standard requires *an* argument
  - Otherwise `COUNT(column)` counts the number of non-null values
    - (We are nice to you and don't make you deal with missing data

Aggregations are Powerful & Common!

```
SELECT date_trunc('day', created) as date, COUNT(*)
FROM users
WHERE created > current_date - interval '1 year'
GROUP BY date;
```

date	count
Apr 17, 2023, 12:00 AM	136
Apr 18, 2023, 12:00 AM	257
Apr 19, 2023, 12:00 AM	326
Apr 20, 2023, 12:00 AM	167
Apr 21, 2023, 12:00 AM	144

Making Groups

- `GROUP BY` allows us to make 'temporary' groups of data.
- Each group becomes a *row* in the final table.

```
SELECT cashier FROM sales GROUP BY cashier
SELECT COUNT(*), cashier FROM sales GROUP BY
cashier
```

id	cashier	cone_id
1	Baskin	2
2	Robin	3
3	Baskin	1
4	Baskin	2
5	Robin	2
6	Robin	1

id	cashier	cone_id
1	Baskin	2
3	Baskin	1
4	Baskin	2

id	cashier	cone_id
2	Robin	3
5	Robin	2
6	Robin	1

COUNT(*)	cashier
3	Baskin
3	Robin

Filtering on Groups

- `GROUP BY` allows us to make 'temporary' groups of data
- HAVING applies a filter after the groups have been made

```
SELECT COUNT(*), flavor FROM sales s JOIN cones c
ON s.cone_id = c.id GROUP BY cone_id HAVING
COUNT(*) > 2
```

id	cashier	cone_id
1	Baskin	2
2	Robin	3
3	Baskin	1
4	Baskin	2
5	Robin	2
6	Robin	1

id	cashier	cone_id
3	Baskin	1
6	Robin	1

id	cashier	cone_id
1	Baskin	2
4	Baskin	2
5	Robin	2

id	cashier	cone_id
2	Robin	3

COUNT(*)	flavor
3	Chocolate

Putting It All Together: JOINs and Aggregations

- Which of our cashiers sold the highest value of ice cream?
- First we need to find which cones were sold by whom, then we SUM() the results!

```
sqlite> SELECT ______
FROM cones c JOIN sales s ON __________
GROUP BY ________________
Cashier|Total Sold
Baskin|13.3
Robin|13.8
```

Michael Ball | UC Berkeley | https://c88c.org | © CC BY-NC-SA

Putting It All Together: JOINs and Aggregations

- Which of our cashiers sold the highest value of ice cream?
- First we need to find which cones were sold by whom, then we SUM() the results!

```
sqlite> SELECT Cashier, SUM(Price) as 'Total Sold'
FROM cones c JOIN sales s ON s.cone_id = c.id
GROUP BY Cashier
Cashier|Total Sold
Baskin|13.3
Robin|13.8
```

Bonus -- billboard hot 100 data

- Download 24.zip and open billboard.db
- All weekly Hot 100 Chart data since the 1950's through 2024.

```
sqlite> .schema
CREATE TABLE hot_100 (
    week_ending text,
    rank integer,
    title text,
    artist text,
    image_url text,
    peak_position integer,
    weeks_on_chart integer
);
```

```
CREATE TABLE tiktok_top_50 (
    week_ending text,
    rank integer,
    title text,
    artist text,
    image_url text,
    peak_position integer,
    weeks_on_chart integer
);
```

 © CC BY-NC-SA

Computational Structures in Data Science

SQL: CREATE and INSERT and UPDATE

(THIS IS NOT TESTED IN 88C!)

UC Berkeley

CC BY-NC-SA

CREATE TABLE

- SQL often used interactively
  - Result of select displayed to the user, but not stored
- [Can create a table in many ways]
  - Often may just supply a list of columns without data.
- Create table statement gives the result a name
  - Like a variable, but for a permanent object

```
CREATE TABLE [name] AS [select statement];
```

SQL: creating a named table

```
CREATE TABLE cones AS
    select 1 as ID, "strawberry" as Flavor, "pink" as Color,
3.55 as Price union
    select 2, "chocolate", "light brown", 4.75 union
    select 3, "chocolate", "dark brown", 5.25 union
    select 4, "strawberry", "pink",5.25 union
    select 5, "bubblegum", "pink",4.75 union
    select 6, "chocolate", "dark brown", 5.25;
```

Notice how column names are introduced and implicit later on.

Inserting new records (rows)

- A database table is typically a shared, durable repository shared by multiple applications

```
INSERT INTO table(column1, column2,...)
        VALUES (value1, value2,...);
```

```
sqlite> insert into cones(ID, Flavor, Color, Price) values (7, "Vanila", "White", 3.95);
sqlite> select * from cones;
ID|Flavor|Color|Price
1|strawberry|pink|3.55
2|chocolate|light brown|4.75
3|chocolate|dark brown|5.25
4|strawberry|pink|5.25
5|bubblegum|pink|4.75
6|chocolate|dark brown|5.25
7|Vanila|White|3.95
sqlite>
```

```
cones.append((7, "Vanila", "White", 3.95))
cones
```

ID	Flavor	Color	Price
1	strawberry	pink	3.55
2	chocolate	light brown	4.75
3	chocolate	dark brown	5.25
4	strawberry	pink	5.25
5	bubblegum	pink	4.75
6	chocolate	dark brown	5.25
7	Vanila	White	3.95

UPDATING new records (rows)

- If you don't specify a WHERE, you'll update all rows!

```
UPDATE table SET column1 = value1, column2 =
value2 [WHERE condition];
```

Summary

```
SELECT  <col spec> FROM <table spec> WHERE <cond spec>
    GROUP BY <group spec> ORDER BY <order spec> ;
```

```
INSERT INTO table(column1, column2,...)
      VALUES (value1, value2,...);
```

```
CREATE TABLE name ( <columns> ) ;
```

```
CREATE TABLE name AS  <select statement> ;
```

© CC BY-NC-SA