Computational Structures in Data Science

Filtering in SQL

UC Berkeley

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

Filtering rows - where

- Set of Table records (rows) that satisfy a condition

```
select [columns] from [table] where [condition] order by [order] ;
```

```
In [5]: cones.select(['Flavor', 'Price'])
Out[5]:
```

Flavor	Price
strawberry	3.55
chocolate	4.75
chocolate	5.25
strawberry	5.25
bubblegum	4.75
chocolate	5.25

```
sqlite> select * from cones where Flavor = "chocolate";
ID|Flavor|Color|Price
2|chocolate|light brown|4.75
3|chocolate|dark brown|5.25
6|chocolate|dark brown|5.25
```

```
cones.where(cones["Price"] > 5)
```

ID	Flavor	Color	Price
3	chocolate	dark brown	5.25
4	strawberry	pink	5.25
6	chocolate	dark brown	5.25

SQL:

```
sqlite> select * from cones where Price > 5;
ID|Flavor|Color|Price
3|chocolate|dark brown|5.25
4|strawberry|pink|5.25
6|chocolate|dark brown|5.25
```

SQL Operators for predicate

- use the WHERE clause in the SQL statements such as SELECT, UPDATE and DELETE to filter rows that do not meet a specified condition

SQLite understands the following binary operators, in order from highest to lowest precedence:

```
||
*    /    %
+    -
<<   >>   &    |
<    <=   >    >=
=    ==   !=   <>   IS   IS NOT   IN   LIKE   GLOB   MATCH   REGEXP
AND
OR
```

Supported unary prefix operators are these:

```
-    +    ~    NOT
```

© CC BY-NC-SA

Approximate Matching: LIKE [Docs]

- LIKE compares text to a *pattern*
  - *Case-Insensitive* by default. Means 'a' and 'A' are the same.
- Allows "wildcards" that match any character.
- % means "zero or more" characters at this "spot" in the pattern
- Examples:

```
'abc' LIKE 'abc' → true
'abc' LIKE 'a%' → true
'abc' LIKE '%b%' → true -shortcut for "does abc contain b?"
'b' LIKE '%b%'  → true
'abc' LIKE 'c' → false
```

 © CC BY-NC-SA

Practice

- How do we find all 'brown' colored ice cream flavors?

```
SELECT *
FROM cones
WHERE ______________________________________;
```

- How we find all 'berry' flavors?

```
SELECT *
FROM cones
WHERE ______________________________________;
```

Practice

- How do we find all 'brown' colored ice cream flavors?

```
SELECT *
FROM cones
WHERE color LIKE '%brown';
```

- How we find all 'berry' flavors?

```
SELECT *
FROM cones
WHERE flavor LIKE '%berry';
```

© CC BY-NC-SA

Computational Structures in Data Science

SQL: Aggregations

UC Berkeley

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

© 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
```

 © CC BY-NC-SA

Unique & DISTINCT values

```
select DISTINCT [columns] from [table] where [condition] order by [order];
```

```
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

Computational Structures in Data Science

SQL: Joins

UC Berkeley

Dealing with Multiple Tables

- Databases often consist of many tables
- Each table organizes all data about a single concept
- Let's explore a new table called sales

```
sqlite> SELECT * FROM sales;
Cashier|id|cone_id
Baskin|1|2
Baskin|3|1
Baskin|4|2
Robin|2|3
Robin|5|2
Robin|6|1
```

- This table tracks a sale of ice cream cones, namely who sold it and what ice cream cone was sold
- Tables tend not to duplicate information

© CC BY-NC-SA

Joining tables

- `JOIN`ing allows combining data from multiple tables
- We have cones and sales? How do we know much each cashier sold? The sales table doesn't contain prices!
- Two tables are joined by a comma to yield all combinations of a rows, the Cartesian Product

```
SELECT * FROM cones, sales;
```

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

© CC BY-NC-SA

Joins

- A "cross product" or full join gives *all combinations*
- **This is often not useful!**
- How do we know which sales record is linked to which cone?
- 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
- Solution: Filter the results!

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

```
SELECT * FROM cones JOIN sales ON sales.cone_id = cones.id
```

Inner Join

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

```
SELECT * FROM cones JOIN sales ON sales.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
```

How Many Sales?

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

```
SELECT * FROM cones JOIN sales ON sales.cone_id = cones.id
```

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

```
sqlite> SELECT COUNT(*), Cashier FROM cones, sales WHERE
cone_id = cones.id GROUP BY cashier;
3|Baskin
3|Robin
sqlite>
```

Putting It All Together:

- 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 _______, _________ as 'Total Sold'
FROM cone
JOIN ____________________________
GROUP BY ____________________________
Cashier|Total Sold
Baskin|13.3
Robin|13.8
```

Putting It All Together:

- 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 sales, cones
WHERE sales.cone_id = cones.id
GROUP BY cashier;
Cashier|Total Sold
Baskin|13.3
Robin|13.8
```

© CC BY-NC-SA

Putting It All Together:

- 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
JOIN sales ON sales.cone_id = cones.id
GROUP BY cashier;
Cashier|Total Sold
Baskin|13.3
Robin|13.8
```

Queries within queries

- Any place that a table is named within a select statement, a table could be computed
  - As a sub-query

```
select TID from sales where Cashier is "Baskin";

select * from cones
    where ID in (select TID from sales where Cashier is "Baskin");

sqlite> select * from cones
   ...>     where ID in (select TID from sales where Cashier is "Baskin");
ID|Flavor|Color|Price
1|strawberry|pink|3.55
3|chocolate|dark brown|5.25
4|strawberry|pink|5.25
```

© CC BY-NC-SA

Computational Structures in Data Science

SQL: Data Manipulation Language

CREATE and INSERT and UPDATE

UC Berkeley

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] );
```

```
CREATE TABLE [name] (
    column1 datatype, column2 datatype
);
```

© CC BY-NC-SA

SQL: creating a named table

Notice how column names are introduced and implicit later on.

```
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;
```

```
CREATE TABLE cones (
    id INTEGER,
    flavor TEXT,
    price REAL -- A decimal type
);
```

© CC BY-NC-SA

INSERT INTO

INSERT in SQL matches tuples of column names to values.

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

```
INSERT INTO table(column1, column2,...)
VALUES (value1, value2, ...), -- Record 1
       (value1, value2, ...), -- Record 2
       (value1, value2, ...);
```

```
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

```
INSERT INTO cones(id, flavor, color, price)
VALUES (7, ‘vanilla’, ‘white’, 3.95);
```

UPDATE

Set all matching column values to the result of a new expression

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

```
-- Increase the cost of ALL cones
UPDATE cones SET
  price = price + 0.5;

-- Oh no, berries are expensive!
UPDATE cones SET
  price = price + 1
  WHERE flavor LIKE ‘%berry%’;
```

Computational Structures in Data Science

Summary

UC Berkeley

Summary — Selecting

```
SELECT <column expressions>
FROM <tables>
WHERE <cond spec> ;
```

```
SELECT <column expressions>
FROM <tables>
JOIN <tables> ON <column conditions>
WHERE <cond spec>
GROUP BY <group spec>
ORDER BY <order spec> [ASC | DESC];
```

Summary — Manipulating Data

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

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

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

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

 © CC BY-NC-SA