NYU | Center for Data Science

Week 02.2: SQL

DS-GA 1004: Big Data

This week

- Relational databases
- **SQL**
- Transactions

Structured Query Language

- SQL is the language we use to talk to databases
  - Not a procedural language like Python or C
  - **Declarative**: state what you want, not how to compute it
- Think of it more like a **protocol** than a programming language
- SQL is an ANSI standard, but different implementations each have quirks
  - **MySQL** vs **Postgres** vs **SQLite** vs **MSSQL** …

SELECTing data

- Get all rows: **SELECT** * **FROM** Dinosaur

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False
4	Homer	Boomer	Donuts	False

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False
4	Homer	Boomer	Donuts	False

SELECTing data

- Get all rows: **SELECT** * **FROM** Dinosaur
- Get some rows: **SELECT** * **FROM** Dinosaur **WHERE** Awesome = True

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False
4	Homer	Boomer	Donuts	False

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True

SELECTing data

- Get all rows: **SELECT** * **FROM** Dinosaur
- Get some rows: **SELECT** * **FROM** Dinosaur **WHERE** Awesome = True
- Get columns: **SELECT** Era, Species **FROM** Dinosaur **WHERE** id > 2

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False
4	Homer	Boomer	Donuts	False

Era	Species
Cretaceous	Ankylosaurus
Boomer	Homer

Selection

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False
4	Homer	Boomer	Donuts	False

- Remove tuples by filtering (WHERE …)
- And remove / rename / reorder columns
- Result of SELECT is always another relation
- You typically iterate over rows produced by SELECT in your host language

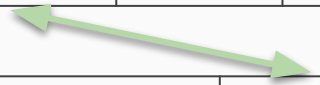
```
for row in db.execute('SELECT * FROM Dinosaurs'):
    print(row)
```

Python + sqlite3 example

Joining relations

- Data is typically structured across multiple relations
- We can combine relations by **JOIN**ing
- **SELECT** * from Dinosaur **JOIN** Character

id	Species	Era	Diet	Awesome
1	T. Rex	Cretaceous	Carnivore	True
2	Stegosaurus	Jurassic	Herbivore	True
3	Ankylosaurus	Cretaceous	Herbivore	False



id	Name	Species	Internals
1	Earl Sinclair	Megalosaurus	Puppet
2	Grimlock	T. Rex	Robot
3	Snarl	Stegosaurus	Robot

A [?] JOIN B

Least specific

Most specific

CROSS JOIN	**All combinations** of rows (r_1, r_2) $r_1 \in A$, $r_2 \in B \Rightarrow A \times B$ (no matching condition)
[LEFT/RIGHT/FULL] OUTER JOIN	**All rows are retained** from A (**LEFT**) or B (**RIGHT**), even if no match is found. Fill missing data with **NULL**
INNER JOIN	**Only matching rows** are retained (Like **OUTER** but without NULLs)
NATURAL JOIN	Rows must match on **all shared columns** (Special case of **INNER**)

A [?] JOIN B

INNER and **OUTER** joins are most common

Least specific

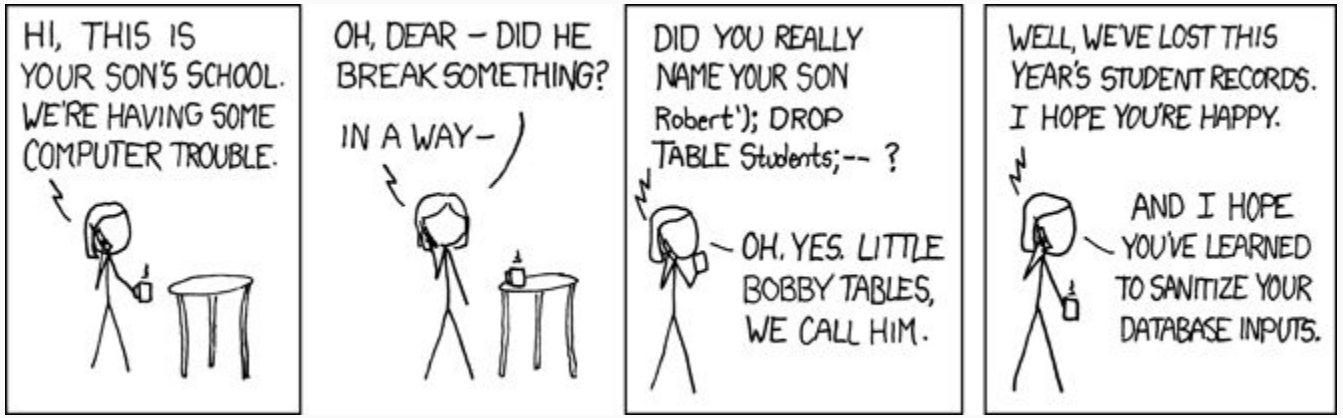
Most specific

CROSS JOIN	**All combinations** of rows (r_1, r_2) $r_1 \in A, r_2 \in B \Rightarrow A \times B$ (no matching condition)
[LEFT/RIGHT/FULL] OUTER JOIN	**All rows are retained** from A (**LEFT**) or B (**RIGHT**), even if no match is found. Fill missing data with **NULL**
INNER JOIN	**Only matching rows** are retained (Like **OUTER** but without NULLs)
NATURAL JOIN	Rows must match on **all shared columns** (Special case of **INNER**)

Modifying data

INSERT INTO table (column1, column2, …)
VALUES (value1, value2, …),
[(row_2_value1, row_2_value2, …), ...]

UPDATE table
SET column1 = value1, column2 = value2, …
WHERE [some condition]



https://xkcd.com/327/

Use your RDBMS library to sanitize queries

```
db.execute(“ SELECT * FROM Dinosaur
             WHERE species = ‘%s’” % name)
```

name variable becomes part of the query code!

```
db.execute(“ SELECT * FROM Dinosaur
             WHERE species = ‘?’”, name)
```

name variable is a **parameter** to the query code!

Python + sqlite3 example

Aggregation

Aggregation queries

- Aggregation lets us summarize multiple tuples into a single result
- **Example**: find the average height of people within a zip code

SELECT Zip, AVG(Height) **FROM** Residents **GROUP BY** Zip

Zip	AVG(Height)
10003	2.68
10004	2

id	Name	Height	Street	Zip
1	T. Rex	3.66	5th Ave.	10003
2	Stegosaurus	2	8th St.	10004
3	Ankylosaurus	1.7	Lafayette St.	10003

Some useful aggregators

- **AVG, SUM, MIN, MAX** ⇐ what you expect
- **COUNT**(**DISTINCT** x) ⇐ # of unique values of column x
- **COUNT**(*) vs **COUNT**(x) ⇐ # rows vs # non-nulls of a column
- **GROUP_CONCAT**(x) ⇐ concatenate (string) values
  GROUP_CONCAT(x, y) ⇐ same, but join with string y

Aggregation conditions

- **SELECT** … **WHERE** [condition] **GROUP BY** [fields]
- **WHERE** clause applies to input, not **output**
- What if you only want to keep certain groups (e.g., sum > 10)?
  - … **HAVING** [group condition]

Aggregation conditions

- **SELECT** … **WHERE** [condition] **GROUP BY** [fields]
- **WHERE** clause applies to input, not **output**
- What if you only want to keep certain groups (e.g., sum > 10)?
  - … **HAVING** [group condition]
  - **SELECT** sum(Height) **FROM** TallDinos **GROUP BY** zip **HAVING** sum(Height) > 10

Indexing

Logical and Physical storage

- Relational schemas provide one view of the data
  - Set or list of **tuples**
- This may not be the best way to organize the data internally
  - Organizing by column can be much more efficient!
  - And what data structure do you use for each type? Hash tables? Trees?
- RDBMS abstract these decisions away from you (the user)
  - But sometimes you can help it out, if you know how data will be used

Indexing

- **index**: a data structure over one or more columns that can accelerate queries
- **Example**:
  - A table that has a few distinct values repeated millions of times
  - And you frequently want all rows with exactly one given value
  - It might be faster to store a mapping **value → rows**
    than to search each row independently

id	Name	Country	Street	Zip
1	T. Rex	US	5th Ave.	10003
2	Stegosaurus	US	8th St.	10004
3	Ankylosaurus	CA	Spadina Ave.	M5T 3A5

Drawbacks of indices

- They take time and **space** to construct
- **Composite indices** (multiple columns) are particularly costly
- **Updates** become slower
- No guarantee that they will help in all queries

When to index?

- When data is **read more often** than written
- When queries are **predictable**
- When queries rely on a **small number of attributes**
- **Remember**: you can always add or delete indices later

Summary

SQL is magic!

- SQL provides a standard interface to relational databases
- Modern frameworks often provide SQL-style interfaces
  - Pandas .groupBy(), .merge(), etc…
- Use indices to organize your data ahead of time