Week 02: SQL

Data Science Bootcamp
Summer, 2021

Instructor: Sagar Patel

Where are we?

Week 1 — Week 2 — Week 3 — Week 4 — Week 5 — Week 6

Week 1	Week 2	Week 3	Week 4	Week 5	Week 6
Introduction	**SQL**	**Python libraries**	**Exploratory Data Analysis**	**Machine Learning**	**Project & Wrap**
Introduction to Python; Getting started with Git and GitHub					

Communities

- Join the **Slack** community to not miss out on any announcements and updates
  Link: https://join.slack.com/t/nyu-dsbc-s21/shared_invite/zt-reskrjo0-eCwFfCmnVYnTAc82lTVNJw
- Share your **GitHub** Username on **#general** to be added to the NYU Data Science Bootcamp Organization where all the resources and tasks will be available after each session
  - If you do not have a GitHub account, create one!
- You can also email us at **datasciencebootcamp@nyu.edu**

Agenda

- Database Management Systems
  - Relational Databases
- Structured Query Language (SQL)
  - **SELECT**ing, **JOIN**ing, and modifying data
  - Aggregations
  - Indexing

NOTE: Database research is a huge topic!

This intro will be brief and cursory.

Database

slido

Have you worked with Databases before?

ⓘ Start presenting to display the poll results on this slide.

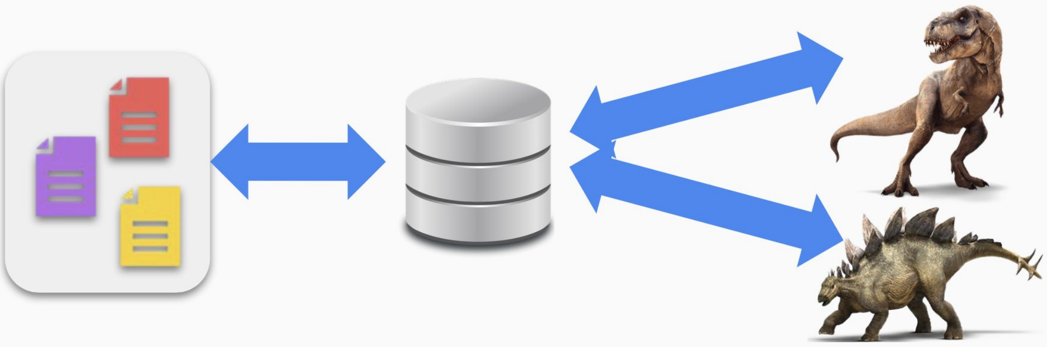
What is a Database?

- An **organized collection** of structured information, or data, typically stored electronically in a computer system.
- What happens if the information is stored on **cloud**? Is that a database?
  - **Yes**, it is.
  - The information stored is still an organized collection, even if it is not stored in the same computer system.

Database management systems (DBMS)

- DBMS's job is to provide
  - Data integrity / consistency
  - Concurrent access
  - Efficient storage and access
  - Standardized format / administration
  - Standardized query interface (language)

- DBMS come in many flavors
  - **Relational (RDBMS)**
  - Semi-structured (e.g., XML)
  - Object-oriented
  - Object-relational
  - ...



The relational model

- **High-level**: tables of data that you are probably used to
  - Spreadsheets, dataframes, numerical arrays, etc.
- Each column represents a **set** of possible values (numbers, strings, etc.)

The relational model

- **High-level**: tables of data that you are probably used to
  - Spreadsheets, dataframes, numerical arrays, etc.
- Each column represents a **set** of possible values (numbers, strings, etc.)

- A ***relation*** over sets $A_1, A_2, A_3, \ldots, A_n$ is a **subset** of their cartesian product
  - $R \subseteq A_1 \times A_2 \times A_3 \times \ldots \times A_n$
  - The **rows** of the table are elements of **R**, also known as ***tuples***
  - $(a_1, a_2, \ldots, a_n) \in R \Rightarrow a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n$

Example: dinosaurs

- A_1 = {s | s is a **string**}
  A_2 = {“Jurassic”, “Cretaceous”, “Devonian”, “Triassic”, ...}
  A_3 = {“Carnivore”, “Herbivore”, “Omnivore”, ...}
  A_4 = {False, True}
- Any A_i could be finite or infinite
- $R \subseteq A_1 \times A_2 \times A_3 \times A_4$ need not contain all combinations!

Species	Era	Diet	Awesome
T. Rex	Cretaceous	Carnivore	True
Stegosaurus	Jurassic	Herbivore	True
Ankylosaurus	Cretaceous	Herbivore	False

Aside: Why “relations” and not “tables”?

- Relations are the abstract model of data
- **Table** refers to an **explicitly** constructed relation
  - i.e., records you have observed / collected
- Other relations in a database:
  - **View**: A relation defined **implicitly**, and constructed **dynamically** at run-time
  - **Temporary table**: the output of a **query**

Properties of relations

- **R** $\subseteq$ $A_1 \times A_2 \times A_3 \times \ldots \times A_n$ is a set
  - The tuples (rows) of **R** are **unordered**
  - Tuples are **unique** ⇒ no duplicates!
  - Relations over common domains (columns) can be combined by set operations

Species	Era	Diet	Awesome
T. Rex	Cretaceous	Carnivore	True
Stegosaurus	Jurassic	Herbivore	True
Ankylosaurus	Cretaceous	Herbivore	False

Properties of relations

- $R \subseteq A_1 \times A_2 \times A_3 \times \dots \times A_n$ is a set
  - The tuples (rows) of **R** are **unordered**
  - Tuples are **unique** ⇒ no duplicates!
  - Relations over common domains (columns) can be combined by set operations
- In practice, add a column (e.g., A_0) with **identifiers** to force uniqueness
  - This is not (usually) part of the data, but is generated automatically by the DBMS
  - ID fields are often used as **primary keys**, and give a default order to rows

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

Schemas

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

- A relation is defined by a **schema**:

Dinosaur(id: int, Species: string, Era: string, Diet: string, Awesome: boolean)

- **Any** tuple (int, string, string, string, boolean) is valid under this schema
  - Schemas enforce type (syntax), but not semantics!

Relational databases

- A relational database consists of one or more relational schemas
- Structured data can be encoded by **joining** on **shared attributes**
- The collection of schemas defines your **data model**

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

Keys

id	First Name	Last Name	Age
1	Homer	Simpson	39
2	Marge	Simpson	39
3	Bart	Simpson	10
4	Homer	Thompson	39
5	Homer	Simpson	28

- Keys are what determine the **identity** of a row
- Keys can be **simple** (single column) or **compound** (two or more columns)
  - Example: **(First Name, Last Name)**
  - This prevents two rows with the same combination of first and last name

- You can have **primary** and **alternate keys**
  - Usually a good idea to keep a primary numeric key as well as others you may want...

Foreign Keys

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

id	Name	DinosaurID	Internals
1	Earl Sinclair	25	Puppet
2	Grimlock	1	Robot
3	Snarl	3	Robot

- A **key** from one relation can be a **column** in another
  - This is called a **FOREIGN KEY** constraint
- This can be used to ensure reference consistency **between** tables/relations
- **This is not automatic**: must be included in the **schema** definition!

SQL

slido

Do you have any experience working with SQL?

ⓘ Start presenting to display the poll results on this slide.

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** ...

Using CREATE

- While working with an SQL server on local computer:

```
CREATE DATABASE [Name];
SHOW DATABASES;
USE [Name];
```

- Creating a table:

```
CREATE TABLE    Dinosaur (
                id          INT AUTO INCREMENT,
                Species     VARCHAR(20),
                …,
                PRIMARY KEY (id)
);
```

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

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 combination** 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 combination** 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]
```

Aggregation queries

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

- 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

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

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

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

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

- We use relational data everyday without thinking of it
- Database consist of one or more relations
- Schemas provide some degree of safety and validation
- SQL provides a standard interface to relational databases
- Use indices to organize your data ahead of time

That’s all Folks!

See you in the next session :)

Give us a feedback: https://bit.ly/3g6ZDlD