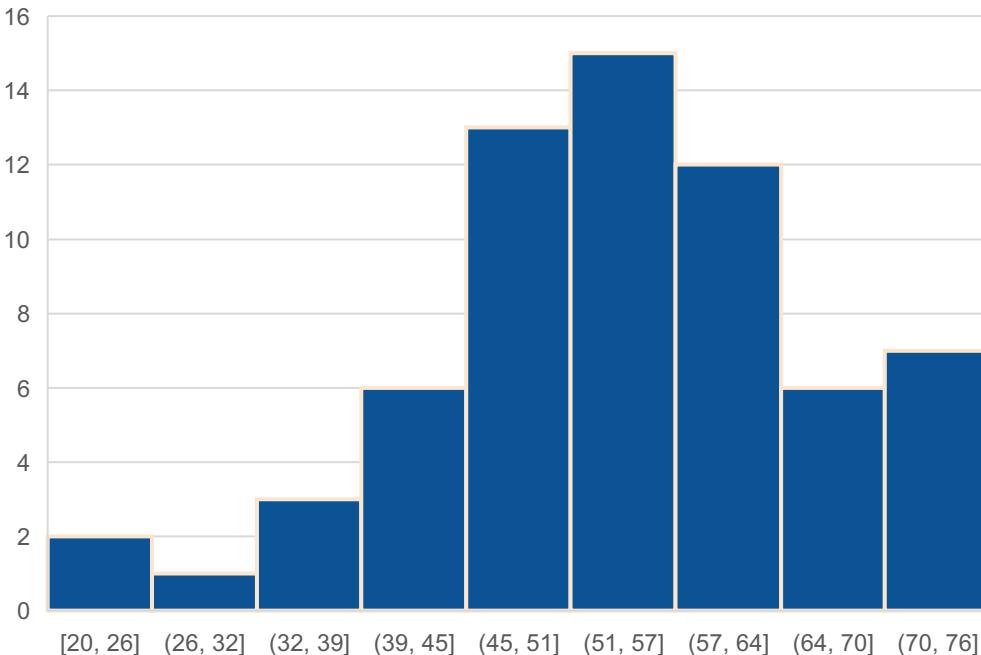


Score



Grade statistics for: Midterm

x

Average Score: **54.66**

High Score: **76.17**

Low Score: **24.5**

Total Graded Submissions: **64 submissions**

LECTURE 20

SQL

SQL and Databases: An alternative to Pandas and CSV files.

In a scale of 0-10, how would you rate your skills in SQL?

0: No background at all!

10: I am an expert!

Go to www.menti.com and use the code 9236 0344

Why Databases

- **Why Databases**
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- DISTINCT
- SQL and Pandas
- LIKE and CAST
- SQL Joins

Previously: CSV Files and Pandas

we've usually worked with data stored in CSV files.

Calls_for_Service.csv

→ pd.read_csv

	CASENO	OFFENSE	EVENTDT	EVENTTM	CVLEGEND	CVDOW	InDbDate	Block_Location	BLKADDR	City	State
0	21014296	THEFT MISD. (UNDER \$950)	04/01/2021 12:00:00 AM	10:58	LARCENY	4	06/15/2021 12:00:00 AM	Berkeley, CA\n(37.869058,-122.270455)	NaN	Berkeley	CA
1	21014391	THEFT MISD. (UNDER \$950)	04/01/2021 12:00:00 AM	10:38	LARCENY	4	06/15/2021 12:00:00 AM	Berkeley, CA\n(37.869058,-122.270455)	NaN	Berkeley	CA
2	21090494	THEFT MISD. (UNDER \$950)	04/19/2021 12:00:00 AM	12:15	LARCENY	1	06/15/2021 12:00:00 AM	2100 BLOCK HASTE ST\nBerkeley, CA\n(37.864908,...)	2100 BLOCK HASTE ST	Berkeley	CA
3	21090204	THEFT FELONY (OVER \$950)	02/13/2021 12:00:00 AM	17:00	LARCENY	6	06/15/2021 12:00:00 AM	2600 BLOCK WARRING ST\nBerkeley, CA\n(37.86393...)	2600 BLOCK WARRING ST	Berkeley	CA
4	21090179	BURGLARY AUTO	02/08/2021 12:00:00 AM	6:20	BURGLARY - VEHICLE	1	06/15/2021 12:00:00 AM	2700 BLOCK GARBER ST\nBerkeley, CA\n(37.86066,...)	2700 BLOCK GARBER ST	Berkeley	CA

Perfectly reasonable workflow for small data that we're not actively sharing with others.

Brief Databases Overview

A **database** is an organized collection of data.

A **database management system (DBMS)** is a software system that **stores, manages**, and **facilitates access** to one or more **databases**.

Why use DBMSes?

- Our data might not be stored in a simple-to-read format such as a CSV (comma-separated values) file.
- Think of a CSV like an Excel sheet.
- Most of the data were given to you in CSV files, but that will not always be the case in the real world.

If our data are stored in a DBMS, we must use languages such as Structured Query Language (SQL) to query for our data.

Advantages of DBMS over CSV (or similar)

Data Storage:

- **Reliable storage** to survive system crashes and disk failures.
- Optimize to **compute on data that does not fit in memory**.
- Special data structures to **improve performance**.

Data Management:

- Configure how data is **logically organized** and **who has access**.
- Can enforce guarantees on the data (e.g. non-negative bank account balance).
 - Can be used to **prevent data anomalies**.
 - Ensures **safe concurrent operations** on data (multiple users reading and writing simultaneously, e.g. ATM transactions).

Warmup: SQL Example

- Why Databases
- **Warmup: SQL Example**
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- DISTINCT
- SQL and Pandas
- LIKE and CAST
- SQL Joins

Today we'll be using a programming language called "Structured Query Language" or SQL.

- SQL is its own programming language, totally distinct from Python.
- SQL is a special purpose programming language used specifically for communicating with databases.
- We will program in SQL using Jupyter notebooks.

Let's see a quick demo of how we can use SQL to connect to a database and view a SQL Table.

Step 1: Loading the SQL Module

Our first step is to load the SQL module. We do so using the ipython magic command below.

```
%load_ext sql
```

Step 2: Connecting to a Database

The next step is to connect the database. We use the `%%sql` header to tell Jupyter that this cell represents SQL code rather than Python code.

```
%%sql  
sqlite:///data/lec20_basic_examples.db
```

Note that in today's example, our database is stored in a local file. In real world practice, you'd probably connect to a remote server, e.g.

```
%%sql  
postgresql://username@mypassword@sjtu.edu.cn/xxx
```

Example: Showing All Tables in the Database

With the command below, we can enumerate all of the tables in the Database.

```
%%sql  
SELECT * FROM sqlite_master WHERE type='table'
```

type	name	tbl_name	rootpage	sql
table	sqlite_sequence	sqlite_sequence	7	CREATE TABLE sqlite_sequence(name,seq)
table	Dragon	Dragon	2	CREATE TABLE Dragon (name TEXT PRIMARY KEY, year INTEGER CHECK (year >= 2000), cute INTEGER)
table	Dish	Dish	4	CREATE TABLE Dish (name TEXT PRIMARY KEY, type TEXT, cost INTEGER CHECK (cost >= 0))
table	Scene	Scene	6	CREATE TABLE Scene (id INTEGER PRIMARY KEY AUTOINCREMENT, biome TEXT NOT NULL, city TEXT NOT NULL, visitors INTEGER CHECK (visitors >= 0), created_at DATETIME DEFAULT (DATETIME('now')))

Example: Showing All Tables in the Database

We see that there are three tables: **Dragon**, **Dish**, and **Scene**.

- The “sql” column gives the command used to create the table.
- Many of the details are beyond the scope of our class. But let’s dig into the most important ideas.

type	name	tbl_name	rootpage	sql
table	sqlite_sequence	sqlite_sequence	7	CREATE TABLE sqlite_sequence(name,seq)
table	Dragon	Dragon	2	CREATE TABLE Dragon (name TEXT PRIMARY KEY, year INTEGER CHECK (year >= 2000), cute INTEGER)
table	Dish	Dish	4	CREATE TABLE Dish (name TEXT PRIMARY KEY, type TEXT, cost INTEGER CHECK (cost >= 0))
table	Scene	Scene	6	CREATE TABLE Scene (id INTEGER PRIMARY KEY AUTOINCREMENT, biome TEXT NOT NULL, city TEXT NOT NULL, visitors INTEGER CHECK (visitors >= 0), created_at DATETIME DEFAULT (DATETIME('now')))

Example: Showing the Dragon Table

Now that we're connected, we can start to make queries.

For example, we might show every row in the "Dragon" table.

```
%%sql  
SELECT * FROM Dragon;
```

The result is displayed in a format almost identical to our Pandas tables (without an index):

	name	year	cute
	hiccup	2010	10
	drogon	2011	-100
	dragon 2	2019	0

SQL Tables

- Why Databases
- Warmup: SQL Example
- **SQL Tables**
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- DISTINCT
- SQL and Pandas
- LIKE and CAST
- SQL Joins

SQL Tables

Let's introduce some key SQL Table terminology by looking at the "Dragon" table.

type	name	tbl_name	rootpage	sql
table	sqlite_sequence	sqlite_sequence	7	CREATE TABLE sqlite_sequence(name,seq)
table	Dragon	Dragon	2	CREATE TABLE Dragon (name TEXT PRIMARY KEY, year INTEGER CHECK (year >= 2000), cute INTEGER)
table	Dish	Dish	4	CREATE TABLE Dish (name TEXT PRIMARY KEY, type TEXT, cost INTEGER CHECK (cost >= 0))
table	Scene	Scene	6	CREATE TABLE Scene (id INTEGER PRIMARY KEY AUTOINCREMENT, biome TEXT NOT NULL, city TEXT NOT NULL, visitors INTEGER CHECK (visitors >= 0), created_at DATETIME DEFAULT (DATETIME('now')))

Column or Attribute or Field

Row or
Record or
Tuple

name TEXT, PK	year INT, >=2000	cute INT
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

Table or Relation

Column or Attribute or Field

Row or
Record or
Tuple



A diagram showing a row in a table. To the left of the row, the text "Row or Record or Tuple" is followed by a large curly brace that spans the width of the row. Above the row, a horizontal brace spans the width of the three columns. The table has three columns with headers: "name" (text, primary key), "year" (int, >=2000), and "cute" (int). The data rows are: "hiccup", "2010", "10"; "drogon", "2011", "-100"; and "dragon 2", "2019", "0".

name TEXT, PK	year INT, >=2000	cute INT
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

Table or Relation

}

Schema:
ColName,
Type,
Constraint

Every column in a SQL table has three properties: **ColName**, **Type**, and zero or more **Constraints**.

- Pandas series have names and types, but no constraints.

Example: Showing All Tables in the Database

Every column in a SQL table has three properties: **ColName**, **Type**, and zero or more **Constraints**.

type	name	tbl_name	rootpage	sql
table	sqlite_sequence	sqlite_sequence	7	CREATE TABLE sqlite_sequence(name,seq)
table	Dragon	Dragon	2	CREATE TABLE Dragon (name TEXT PRIMARY KEY, year INTEGER CHECK (year >= 2000), cute INTEGER)
table	Dish	Dish	4	CREATE TABLE Dish (name TEXT PRIMARY KEY, type TEXT, cost INTEGER CHECK (cost >= 0))
table	Scene	Scene	6	CREATE TABLE Scene (id INTEGER PRIMARY KEY AUTOINCREMENT, biome TEXT NOT NULL, city TEXT NOT NULL, visitors INTEGER CHECK (visitors >= 0), created_at DATETIME DEFAULT (DATETIME('now')))

2 constraints.

Example Types

Some examples of SQL **types**:

- INT: Integers.
- REAL: Real numbers.
- TEXT: Strings of text.
- BLOB: Arbitrary data, e.g. songs, video files, etc.
- DATETIME: A date and time.

Note: Different implementations of SQL support different types.

- sqllite: <https://www.sqlite.org/datatype3.html>
- mysql: <https://dev.mysql.com/doc/refman/8.0/en/data-types.html>

Example Constraints

Some examples of **constraints**:

- CHECK: Data cannot be inserted which violates the given check constraint.
- PRIMARY KEY: Specifies that this key is used to uniquely identify rows in the table.
- NOT NULL: Null data cannot be inserted for this column.
- DEFAULT: Provides a value to use if user does not specify on insertion.

type	name	tbl_name	rootpage	sql
table	sqlite_sequence	sqlite_sequence	7	CREATE TABLE sqlite_sequence(name,seq)
table	Dragon	Dragon	2	CREATE TABLE Dragon (name TEXT PRIMARY KEY, year INTEGER CHECK (year >= 2000), cute INTEGER)
table	Dish	Dish	4	CREATE TABLE Dish (name TEXT PRIMARY KEY, type TEXT, cost INTEGER CHECK (cost >= 0))
table	Scene	Scene	6	CREATE TABLE Scene (id INTEGER PRIMARY KEY AUTOINCREMENT, biome TEXT NOT NULL, city TEXT NOT NULL, visitors INTEGER CHECK (visitors >= 0), created_at DATETIME DEFAULT (DATETIME('now')))

Primary Keys

For the Dragon table, the “name” of each Dragon is the primary key.

- A primary key is used to uniquely identify each record in the table.
- In other words, no two dragons can have the same name!

Primary key is used under the hood for all sorts of optimizations.

name TEXT, PK	year INT, >=2000	cute INT
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

A table can have multiple columns with the PRIMARY KEY **constraints**.

Basic SQL Queries

- Why Databases
- Warmup: SQL Example
- SQL Tables
- **Basic SQL Queries**
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- DISTINCT
- SQL and Pandas
- LIKE and CAST
- SQL Joins

The Simplest Query

Recall our simplest query from before:

```
%%sql  
SELECT * FROM Dragon;
```

	name	year	cute
	hiccup	2010	10
	drogon	2011	-100
	dragon 2	2019	0

Selecting Only Some Columns

We can also SELECT only a subset of the columns:

```
%%sql
```

```
SELECT cute, year FROM Dragon;
```

cute	year
10	2010
-100	2011
0	2019

The AS Keyword

The AS keyword lets us rename columns during the selection process.

```
%%sql
```

```
SELECT cute AS cuteness, year AS birth FROM Dragon;
```

cuteness	birth
10	2010
-100	2011
0	2019

Newline Separators

It is common to use a new line to make SQL code more readable. Compare:

```
%%sql  
SELECT cute AS cuteness, year AS birth FROM Dragon;
```

```
%%sql  
SELECT cute AS cuteness,  
       year AS birth  
FROM Dragon;
```

cuteness	birth
10	2010
-100	2011
0	2019

The WHERE Keyword

To select only some rows of a table, we can use the WHERE keyword.

columns

```
SELECT name, year  
FROM Dragon } table
```

```
WHERE cute > 0;
```

condition

The Row Where Dragon



X X

name	year	cute
------	------	------

hiccup	2010	10
--------	------	----

drogon	2011	-100
--------	------	------

dragon 2	2019	0
----------	------	---

Dragon

name	year
------	------

hiccup	2010
--------	------

Result

The OR Keyword

The OR keyword lets us form a disjunctive condition.

- The AND and NOT keywords also exist.

columns



name	year	cute
------	------	------

hiccup	2010	10
--------	------	----

drogon	2011	-100
--------	------	------

dragon 2	2019	0
----------	------	---

Dragon

```
SELECT name, cute, year
```

```
FROM Dragon } table
```

```
WHERE cute > 0 OR year > 2013;
```

condition

name	cute	year
------	------	------

hiccup	10	2010
--------	----	------

dragon 2	0	2019
----------	---	------

Result

ORDER BY

The ORDER BY keyword lets you sort a Table.

```
all columns
  ^
SELECT *  
FROM Dragon } table  
ORDER BY cute DESC;  
          ^  ^  
        column or ASC
```

name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

name	year	cute
hiccup	2010	10
dragon 2	2019	0
drogon	2011	-100

Result

LIMIT

The LIMIT keyword lets you retrieve N rows.

- Example below is similar to `.head(2)` in pandas.
- Unless you use ORDER BY, there is no guaranteed order!

all columns



`SELECT *`

`FROM Dragon }` table

`LIMIT 2;`



number of
result rows

name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

name	year	cute
hiccup	2010	10
drogon	2011	-100

Result

OFFSET

The OFFSET keyword lets you tell SQL to see later rows when limiting.

- Unless you use ORDER BY, there is no guaranteed order!

all columns



SELECT *

FROM Dragon } table

LIMIT 2

OFFSET 1;



where to start

name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

name	year	cute
drogon	2011	-100
dragon 2	2019	0

Result

Summary So Far

```
SELECT <column list>
FROM <table>
[ WHERE <predicate> ]
[ ORDER BY <column list> ]
[ LIMIT <number of rows> ]
[ OFFSET <number of rows> ];
```

Basic GROUP BY Operations

- Why Databases
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- **Basic GROUP BY Operations**
- Trickier GROUP BY Operations
- DISTINCT
- SQL and Pandas
- LIKE and CAST
- SQL Joins

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

SELECT type

FROM Dish;

type

entree

entree

entree

appetizer

appetizer

appetizer

dessert

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

GROUP BY (similar to groupby in Pandas)

SELECT type

FROM Dish

GROUP BY type;

type

appetizer

dessert

entree

Dish

name <small>TEXT, PK</small>	type <small>TEXT</small>	cost <small>INTEGER, >=0</small>
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Using GROUP BY and MAX (similar to groupby().max() in pandas)

```
SELECT type, MAX(cost)  
FROM Dish  
GROUP BY type;
```

type MAX(cost)

appetizer	4
dessert	5
entree	10

Note that “**type, MAX(cost)**” is called a “column expression list”.

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Using GROUP BY and SUM (similar to groupby().sum() in pandas)

```
SELECT type, SUM(cost)
FROM Dish
GROUP BY type;
```

type SUM(cost)

appetizer	12
dessert	5
entree	24

For more aggregation functions see
https://www.sqlite.org/lang_aggfunc.html

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

```
SELECT type,  
       SUM(cost),  
       MIN(cost),  
       MAX(name)  
FROM Dish  
GROUP BY type;
```

What do you think will happen?

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Using Multiple Aggregation Functions

```
SELECT type,  
       SUM(cost),  
       MIN(cost),  
       MAX(name)
```

```
FROM Dish
```

```
GROUP BY type;
```

type	SUM(cost)	MIN(cost)	MAX(name)
------	-----------	-----------	-----------

appetizer	12	4	potsticker
-----------	----	---	------------

dessert	5	5	ice cream
---------	---	---	-----------

entree	24	7	taco
--------	----	---	------

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

```
SELECT <column expression list>
FROM <table>
[ WHERE <predicate> ]
[ GROUP BY <column> ]
[ ORDER BY <column list> ]
[ LIMIT <number of rows> ]
[ OFFSET <number of rows> ] ;
```

Trickier GROUP BY Operations

- Why Databases
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- **Trickier GROUP BY Operations**
- DISTINCT
- SQL and Pandas
- LIKE and CAST
- SQL Joins

Using GROUP BY and COUNT (similar to groupby().count() in pandas)

SELECT type, COUNT(cost)

FROM Dish

GROUP BY type;

type COUNT(cost)

appetizer	3
-----------	---

dessert	1
---------	---

entree	3
--------	---

COUNT(c) returns the number of non-null entries in a column c.

Dish

name <small>TEXT, PK</small>	type <small>TEXT</small>	cost <small>INTEGER, >=0</small>
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Using GROUP BY and COUNT(*) (similar to groupby().size() in pandas)

```
SELECT type, COUNT(*)
```

```
FROM Dish
```

```
GROUP BY type;
```

type	COUNT(*)
appetizer	3
dessert	1
entree	3

COUNT(*) returns the number of rows in each group, including rows with nulls.

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

```
SELECT type, cost  
FROM Dish  
GROUP BY type;
```

Implementation dependent:

- In some variants of SQL, this is allowed.
- In some variants, it is a syntax error.

This is considered bad practice.

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4

MS SQL:

Column Dish.cost' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

GROUP BY Multiple Columns

```
SELECT type, cost  
FROM Dish  
GROUP BY type, cost;
```

type	cost
------	------

appetizer	4
-----------	---

dessert	5
---------	---

entree	7
--------	---

entree	10
--------	----

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

GROUP BY Multiple Columns

```
SELECT type, cost  
FROM Dish  
GROUP BY type, cost;
```

type	cost
appetizer	4
dessert	5
entree	10

How would we add a third column giving us the number of rows that match each type/cost tuple?

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
sticker	appetizer	4
ice cream	dessert	5

GROUP BY Multiple Columns

```
SELECT type, cost, COUNT(*) Dish
```

```
FROM Dish
```

```
GROUP BY type, cost;
```

type	cost	COUNT(*)
------	------	----------

appetizer	4	3
-----------	---	---

dessert	5	1
---------	---	---

entree	7	2
--------	---	---

entree	10	1
--------	----	---

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Prelude to Filtering Groups: Counting the Size of Each type Group

```
SELECT type, COUNT(*)  
FROM Dish  
GROUP BY type;
```

type	COUNT(*)
appetizer	3
dessert	1
entree	3

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

```
SELECT type, COUNT(*)  
FROM Dish  
GROUP BY type  
HAVING MAX(cost) < 8;
```

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

```
SELECT type, COUNT(*)  
FROM Dish  
GROUP BY type  
HAVING MAX(cost) < 8;
```

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

```
SELECT type, COUNT(*)
FROM Dish
GROUP BY type
HAVING MAX(cost) < 8;
```

type	COUNT(*)
appetizer	3
dessert	1

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Exercise: Difference Between HAVING and WHERE

```
SELECT type, COUNT(*)  
FROM Dish  
WHERE cost < 8  
GROUP BY type;
```

Try to predict what happens!

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Exercise: Difference Between HAVING and WHERE

```
SELECT type, COUNT(*)  
FROM Dish  
WHERE cost < 8  
GROUP BY type;
```



type	COUNT(*)
appetizer	3
dessert	1
entree	2

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

```
SELECT <column expression list>
FROM <table>
[WHERE <predicate>]
[GROUP BY <column list>]
[HAVING <predicate>]
[ORDER BY <column list>]
[LIMIT <number of rows>]
[OFFSET <number of rows>];
```

Distinct

- Why Databases
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- **DISTINCT**
- SQL and Pandas
- LIKE and CAST
- SQL Joins

SELECT DISTINCT type

FROM Dish

WHERE cost < 9;



type

entree

appetizer

dessert

Dish

name <small>TEXT, PK</small>	type <small>TEXT</small>	cost <small>INTEGER, >=0</small>
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

SELECT DISTINCT type, cost

FROM Dish

WHERE cost < 9;

type cost

entree 7

appetizer 4

dessert 5



✗

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

```
SELECT DISTINCT type, cost Dish  
FROM Dish;
```

type	cost
entree	10
entree	7
appetizer	4
dessert	5

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

DISTINCT vs. GROUP BY

```
SELECT DISTINCT type, cost  
FROM Dish;
```

type	cost
entree	10
entree	7
appetizer	4
dessert	5

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

```
SELECT type, cost
```

```
FROM Dish
```

```
GROUP BY type, cost;
```

type	cost
appetizer	4
dessert	5
entree	7
entree	10

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

To get a tables of unique tuples, we saw we could use SELECT DISTINCT or GROUP BY.

- Better style to use SELECT DISTINCT for unique values (or tuples).
- GROUP BY is primarily intended for use with aggregation operations.
 - I think of the example above as a degenerate use of GROUP BY.

DISTINCT in Column Expressions

```
SELECT type, AVG(DISTINCT cost)
```

```
FROM Dish
```

```
GROUP BY type;
```

type AVG(DISTINCT cost)

appetizer	4.0
-----------	-----

dessert	5.0
---------	-----

entree	8.5
--------	-----

Since 10 and 7 are the only two unique cost values,
the average of these two is 8.5.

Dish

name TEXT, PK	type TEXT	cost INTEGER, >=0
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

```
SELECT [DISTINCT] <column expression list>
FROM <table>
[WHERE <predicate>]
[GROUP BY <column list>]
[HAVING <predicate>]
[ORDER BY <column list>]
[LIMIT <number of rows>]
[OFFSET <number of rows>];
```

Note: Column Expressions may include aggregation functions (**MAX**, **MIN**, etc) and **DISTINCT**.

Next lecture: **FROM <list of tables>**



SQL and Pandas

- Why Databases
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- DISTINCT
- **SQL and Pandas**
- LIKE and CAST
- SQL Joins

We just saw how we could use the %%sql magic command to run SQL queries in a Jupyter notebook.

```
%%sql
```

```
SELECT * FROM Dragon;
```

name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

In this section of lecture, we'll see how to use the sqlalchemy Python library to allow communication between pandas and SQL databases.

Example

```
import sqlalchemy
# create a SQL Alchemy connection to the database
engine = sqlalchemy.create_engine("sqlite:///data/lec20_basic_examples.db")
connection = engine.connect()

pd.read_sql("""
SELECT type, MAX(cost)
FROM Dish
GROUP BY type;""", connection)
```

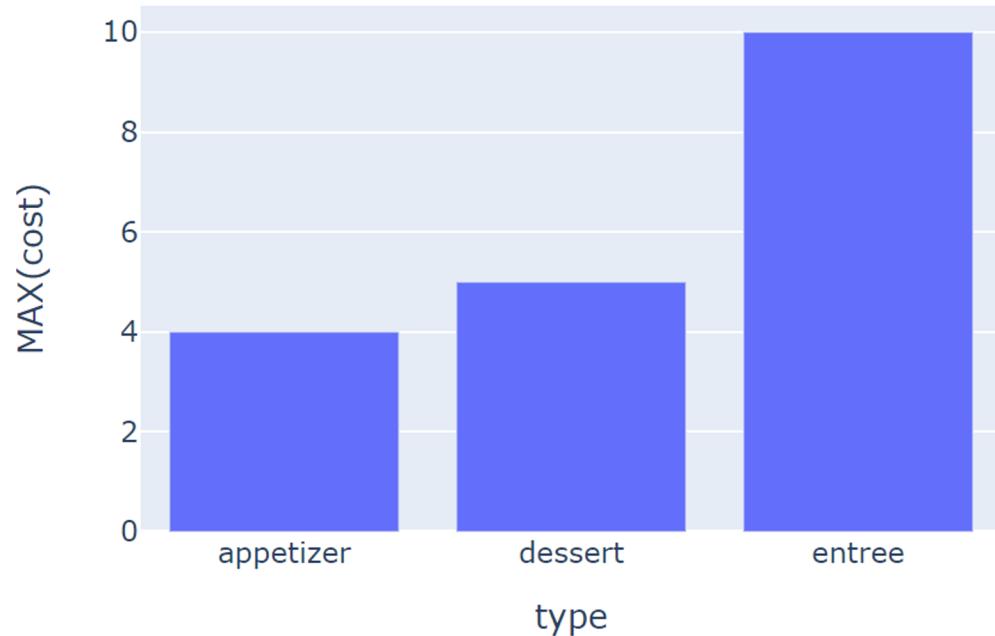
	type	MAX(cost)
0	appetizer	4
1	dessert	5
2	entree	10

Basic idea:

- First create a `sqlalchemy engine`.
- Then call `engine.connect()` to generate a `connection`.
- Then every time you want to make a SQL query, call `pd.read_sql`, providing the **SQL code** as the **first argument**, and the `connection` as the **second argument**.
- Result of `pd.read_sql` is a DataFrame.

Example

```
df = pd.read_sql("""  
SELECT type, MAX(cost)  
FROM Dish  
GROUP BY type;""", connection)  
px.bar(df, x = "type", y = "MAX(cost)")
```



	type	MAX(cost)
0	appetizer	4
1	dessert	5
2	entree	10

LIKE and CAST

- Why Databases
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- DISTINCT
- SQL and Pandas
- SQL and Pandas
- **LIKE and CAST**
- SQL Joins

The IMDB Dataset

IMDB provides a world readable copy of its list of all movies.

- The code below downloads these files, then unzips them.
- The resulting files are stored in tab separated format TSV.
- The titles CSV file is 753 megabytes, too large to be read by pandas on the datahub machines.

```
from os.path import exists

# From https://www.imdb.com/interfaces/
from ds100_utils import fetch_and_cache
data_directory = './data'
fetch_and_cache('https://datasets.imdbws.com/title.basics.tsv.gz',
'titles.tsv.gz', data_directory)
fetch_and_cache('https://datasets.imdbws.com/name.basics.tsv.gz',
'names.tsv.gz', data_directory)
if not exists(f"{data_directory}/titles.tsv"):
    !gunzip -kf {data_directory}/titles.tsv.gz
    !gunzip -kf {data_directory}/names.tsv.gz
```

154112110	Mar	31	10:57	titles.tsv.gz
753395433	Mar	31	10:57	titles.tsv
225456571	Mar	31	10:57	names.tsv.gz
689784115	Mar	31	10:57	names.tsv

The IMDB Dataset

The code below (which we will not describe) converts the tsv files into .db format so that SQL can be used instead.

```
from os.path import exists  
  
imdb_file_exists = exists('./data/imdb.db')  
if not imdb_file_exists:  
    !(cd data; sqlite3 imdb.db ".mode tabs" ".import titles.tsv titles" ".import names.tsv names") 2> /dev/null
```

154112110	Mar 31 10:57	titles.tsv.gz
753395433	Mar 31 10:57	titles.tsv
225456571	Mar 31 10:57	names.tsv.gz
689784115	Mar 31 10:57	names.tsv
665415680	Mar 31 11:21	imdb.db

The IMDB Dataset

To read this file in SQL, we create a sqlalchemy connection as before.

```
engine =  
    sqlalchemy.create_engine("sqlite:///data/imdb.db")  
connection = engine.connect()
```

We can then request the list of Tables:

```
tables = pd.read_sql("SELECT sql FROM sqlite_master WHERE type='table';", connection)  
tables
```

sql

0 CREATE TABLE "titles"(\n "tconst" TEXT,\n "t...

1 CREATE TABLE "names"(\n "nconst" TEXT,\n "pr...

This isn't very readable, so we'll index into the sql column and get rows 0 and 1.

The IMDB Dataset

The two tables are all movies and all actors, respectively, that IMDB tracks.

```
print(tables["sql"][0])
```

```
CREATE TABLE "titles"(  
    "tconst" TEXT,  
    "titleType" TEXT,  
    "primaryTitle" TEXT,  
    "originalTitle" TEXT,  
    "isAdult" TEXT,  
    "startYear" TEXT,  
    "endYear" TEXT,  
    "runtimeMinutes" TEXT,  
    "genres" TEXT  
)
```

```
print(tables["sql"][1])
```

```
CREATE TABLE "names"(  
    "nconst" TEXT,  
    "primaryName" TEXT,  
    "birthYear" TEXT,  
    "deathYear" TEXT,  
    "primaryProfession" TEXT,  
    "knownForTitles" TEXT  
)
```

The IMDB Dataset

The two tables are all movies and all actors, respectively, that IMDB tracks.

```
print(tables["sql"][0])
```

```
CREATE TABLE "titles"(  
    "tconst" TEXT,  
    "titleType" TEXT,  
    "primaryTitle" TEXT,  
    "originalTitle" TEXT,  
    "isAdult" TEXT,  
    "startYear" TEXT,  
    "endYear" TEXT,  
    "runtimeMinutes" TEXT,  
    "genres" TEXT  
)
```

```
print(tables["sql"][1])
```

```
CREATE TABLE "names"(  
    "nconst" TEXT,  
    "primaryName" TEXT,  
    "birthYear" TEXT,  
    "deathYear" TEXT,  
    "primaryProfession" TEXT,  
    "knownForTitles" TEXT  
)
```

Getting 10 Rows

```
get_10_movies = """
SELECT *
FROM titles
LIMIT 10
"""
```

```
pd.read_sql(get_10_movies, conn)
```

	tconst	titleType	primaryTitle	originalTitle	isAdult	startYear	endYear	runtimeMinutes	genres
0	tt0000001	short	Carmencita	Carmencita	0	1894	\N	1	Documentary,Short
1	tt0000002	short	Le clown et ses chiens	Le clown et ses chiens	0	1892	\N	5	Animation,Short
2	tt0000003	short	Pauvre Pierrot	Pauvre Pierrot	0	1892	\N	4	Animation,Comedy,Romance
3	tt0000004	short	Un bon bock	Un bon bock	0	1892	\N	12	Animation,Short
4	tt0000005	short	Blacksmith Scene	Blacksmith Scene	0	1893	\N	1	Comedy,Short
5	tt0000006	short	Chinese Opium Den	Chinese Opium Den	0	1894	\N	1	Short
6	tt0000007	short	Corbett and Courtney Before the Kinetograph	Corbett and Courtney Before the Kinetograph	0	1894	\N	1	Short,Sport
7	tt0000008	short	Edison Kinetoscopic Record of a Sneeze	Edison Kinetoscopic Record of a Sneeze	0	1894	\N	1	Documentary,Short
8	tt0000009	short	Miss Jerry	Miss Jerry	0	1894	\N	40	Romance,Short
9	tt0000010	short	Leaving the Factory	La sortie de l'usine Lumière à Lyon	0	1895	\N	1	Documentary,Short

The LIKE Keyword

The LIKE operator tests whether a string matches a pattern (similar to a regex, but much simpler syntax):

- E.g. select rows where the time string is on the hour, such as 8:00 or 12:00 pm.

```
SELECT * FROM t WHERE t.time LIKE '%:00%';
```

Finding Action Movies

We can use the LIKE keyword to find all Action movies:

```
action_movies_query = f"""
SELECT tconst AS id,
    primaryTitle AS title,
    runtimeMinutes AS time,
    startYear AS year
FROM titles
WHERE titleType = 'movie' AND
    genres LIKE '%Action%'"""
```

```
action_movies = pd.read_sql(action_movies_query, connection)
```

Finding Action Movies

We can use the LIKE keyword to find all Action movies:

```
action_movies_query = f"""
SELECT tconst AS id,
    primaryTitle AS title,
    runtimeMinutes AS time,
    startYear AS year
FROM titles
WHERE titleType = 'movie' AND
    genres LIKE '%Action%'"""

action_movies = pd.read_sql(action_movies_query, engine)
```

IMDB decided to use
"\N" for a missing value.

	id	title	time	year
0	tt0000574	The Story of the Kelly Gang	70	1906
1	tt0003545	Who Will Marry Mary?	\N	1913
2	tt0003747	Cameo Kirby	50	1914
3	tt0003897	The Exploits of Elaine	220	1914
4	tt0004052	The Hazards of Helen	1428	1914
...

Three Problems With Our Data

- We see a number of rows containing "\N". These represent values that IMDB was missing.
- Time and year columns are currently given in string format, whereas we probably want them in numeric format.
- Weird outliers like "The Hazards of Helen" which are 1,428 minutes long.

Could fix in pandas, but let's see how to fix in SQL.

	id	title	time	year
0	tt0000574	The Story of the Kelly Gang	70	1906
1	tt0003545	Who Will Marry Mary?	\N	1913
2	tt0003747	Cameo Kirby	50	1914
3	tt0003897	The Exploits of Elaine	220	1914
4	tt0004052	The Hazards of Helen	1428	1914
...

The **CAST** keyword converts a table column to another type.

- In the code below, we convert runtimeMinutes and startYear to int. Any missing invalid values are replaced by 0 (then the WHERE clauses filter out the 0s).

```
action_movies_query = f'''  
SELECT tconst AS id,  
       primaryTitle AS title,  
       CAST(runtimeMinutes AS int) AS time,  
       CAST(startYear AS int) AS year  
  FROM titles  
 WHERE genres LIKE '%Action%' AND  
       titleType = 'movie' AND  
       time > 60 AND time < 180 AND  
       year > 0  
 ...  
  
action_movies = pd.read_sql(action_movies_query, connection)
```

Result of Our Query

```
action_movies_query = f'''  
SELECT tconst AS id,  
       primaryTitle AS title,  
       CAST(runtimeMinutes AS int) AS time,  
       CAST(startYear AS int) AS year  
FROM titles  
WHERE genres LIKE '%Action%' AND  
      titleType = 'movie' AND  
      time > 60 AND time < 180 AND  
      year > 0  
...  
'''
```

```
action_movies = pd.read_sql(action_movies_query, c
```

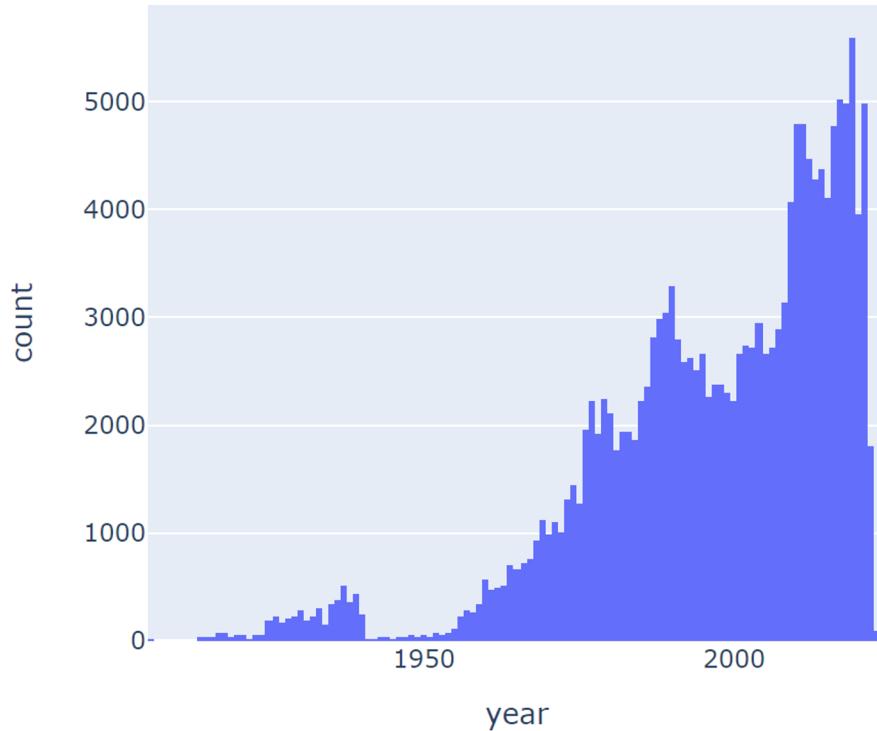
	id	title	time	year
0	tt0000574	The Story of the Kelly Gang	70	1906
1	tt0004223	The Life of General Villa	105	1914
2	tt0004450	Die Pagode	82	1917
3	tt0004635	The Squaw Man	74	1914
4	tt0005073	The Chalice of Courage	65	1915
...
164901	tt9900748	The Robinsons	110	2019
164902	tt9900782	Kaithi	145	2019
164903	tt9900908	Useless Handcuffs	89	1969
164904	tt9901162	The Robinsons	90	2020
164905	tt9904066	Fox Hunting	66	2019

164906 rows × 4 columns

Visualization

Since we have our data in pandas format, we can use our usual visualization tools:

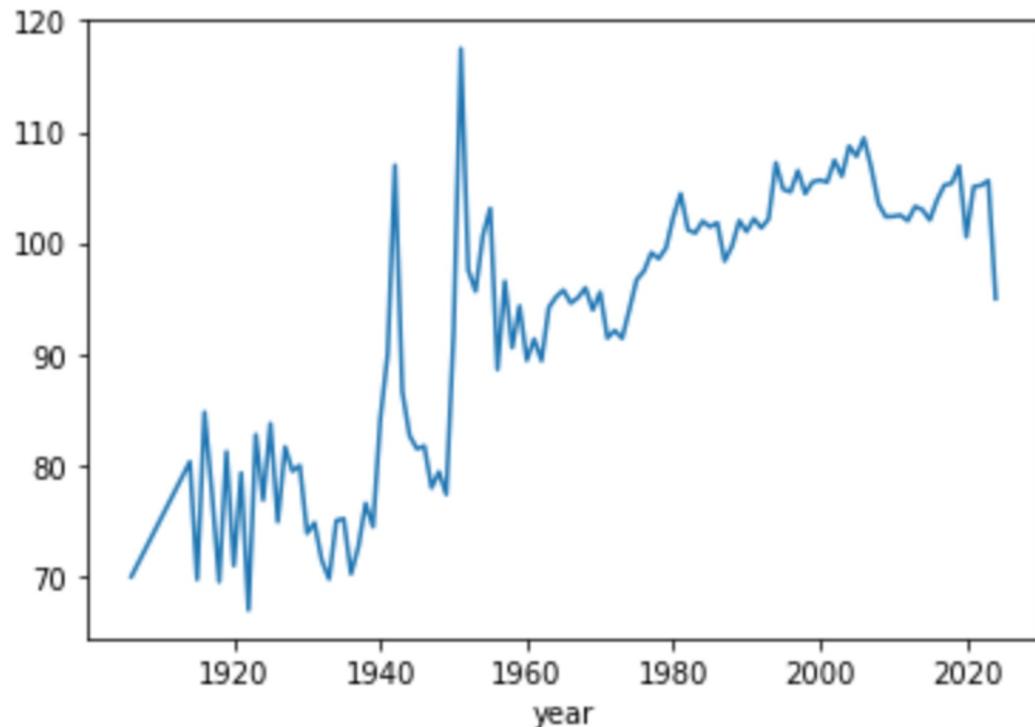
```
px.histogram(action_movies, x = "year")
```



Visualization

Since we have our data in pandas format, we can use our usual visualization tools:

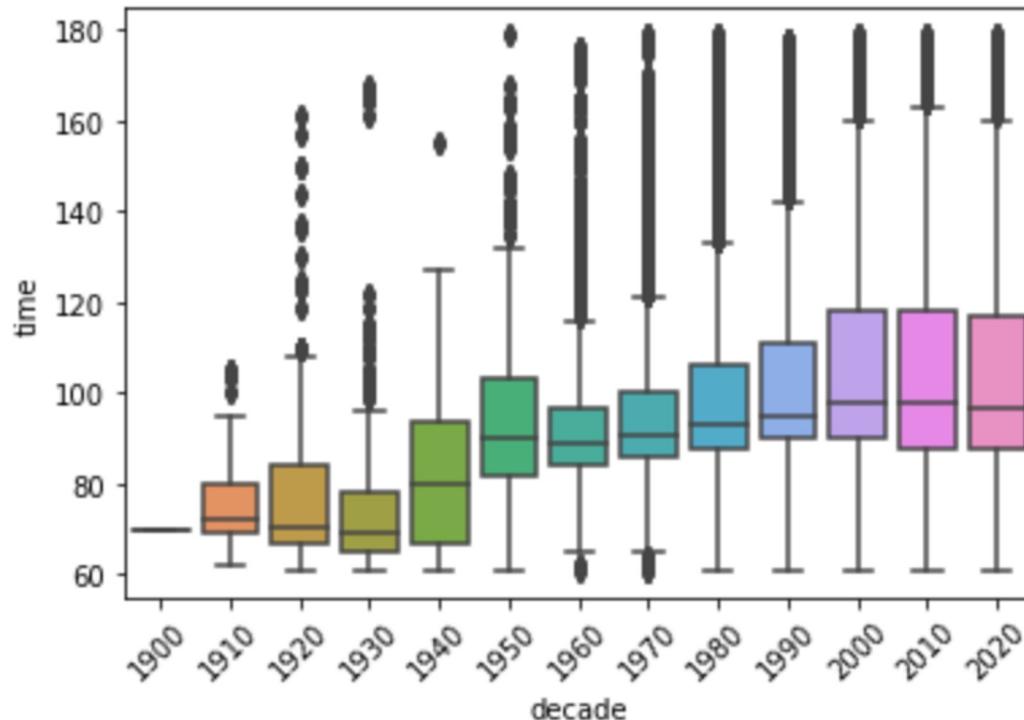
```
action_movies['time'].groupby(action_movies['year']).mean().plot();
```



Visualization

Since we have our data in pandas format, we can use our usual visualization tools:

```
sns.boxplot(x = 'decade', y = 'time', data = action_movies)
```



SQL Joins

- Why Databases
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- DISTINCT
- SQL and Pandas
- LIKE and CAST
- **SQL Joins**

Sales Fact Table

pid	timeid	locid	sales
11	1	1	25
11	2	1	8
11	3	1	15
12	1	1	30
12	2	1	20
12	3	1	50
12	1	1	8
13	2	1	10
13	3	1	10
13	3	2	5

Locations

locid	city	state	country
1	Omaha	Nebraska	USA
2	Seoul		Korea
5	Richmond	Virginia	USA

Products

pid	pname	category	price
11	Corn	Food	25
12	Galaxy 1	Phones	18
13	Peanuts	Food	2

Time

timeid	Date	Day
1	3/30/16	Wed.
2	3/31/16	Thu.
3	4/1/16	Fri.

Real databases are often stored in a format similar to this shown!

Fact table:

- Minimizes redundant info.
- Reduces data errors.

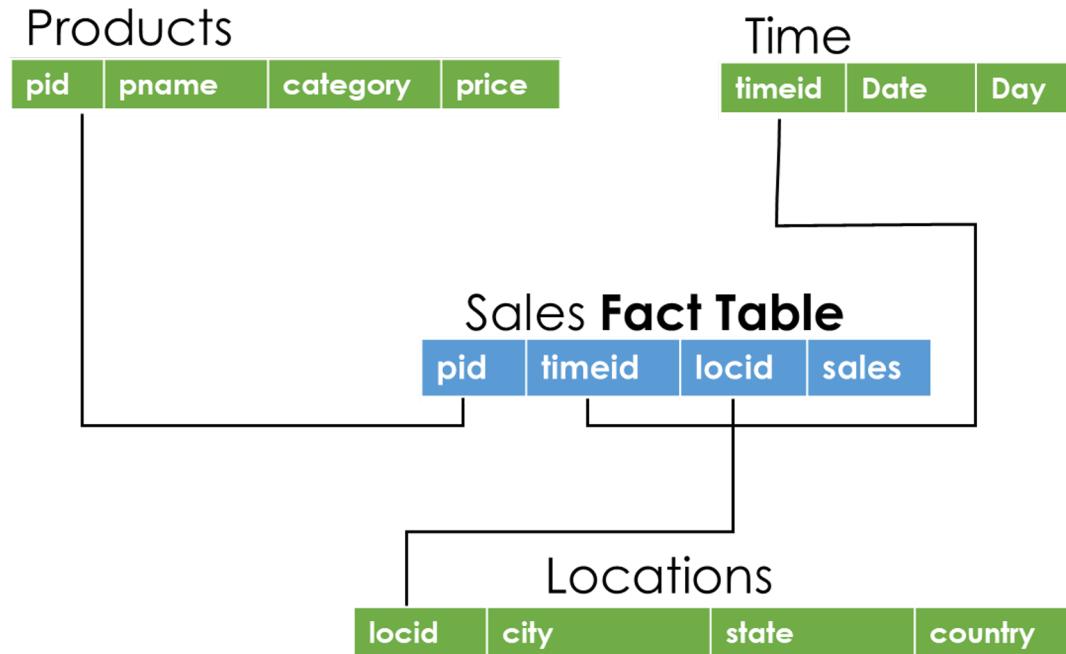
Dimensions:

- Easy to manage and summarize.
- Renaming is easy, just change Galaxy 1 to Phablet.

Connections Between Tables

To do analysis, we'll need to join our tables!

- Aside: This sort of table organization is often called a “star schema”.



Another Inner Join

An example of a SQL join is shown below:

Latitude	Longitude	Name
34	118	Los Angeles
42	71	Cambridge
45	93	Minneapolis

City	Temp
Los Angeles	68
Chicago	59
Minneapolis	55

```
SELECT name, latitude, temp  
FROM cities, temps  
WHERE name = city;
```

Name	Latitude	Temp
Los Angeles	118	68
Minneapolis	93	55

This is an “inner joins”. It turns out there are other types of joins.

Cross Join

In a cross join, all pairs of rows appear in the result!

- This is also known as the Cartesian product.

s	
<u>id</u>	<u>name</u>
0	Apricot
1	Boots
2	Cally
4	Eugene

t	
<u>id</u>	<u>breed</u>
1	persian
2	ragdoll
4	bengal
5	persian

SELECT * FROM s, t;

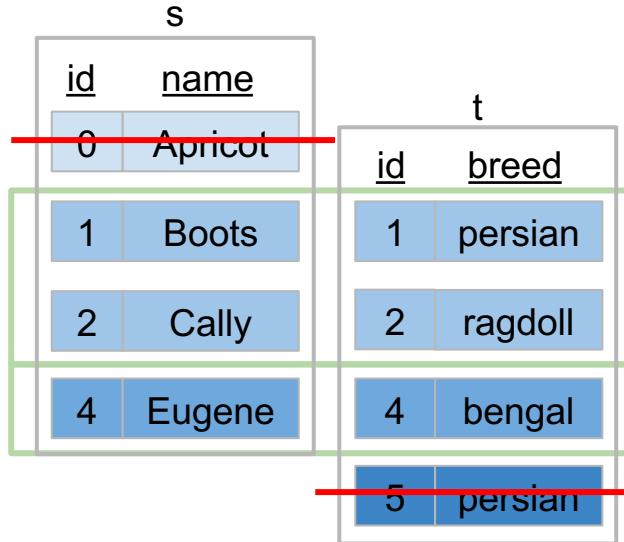
<u>s.id</u>	<u>name</u>	<u>t.id</u>	<u>breed</u>
0	Apricot	1	persian
1	Boots	1	persian
2	Cally	1	persian
4	Eugene	1	persian
0	Apricot	2	ragdoll
1	Boots	2	ragdoll
2	Cally	2	ragdoll
4	Eugene	2	ragdoll

(to be continued ...)

<u>s.id</u>	<u>name</u>	<u>t.id</u>	<u>breed</u>
0	Apricot	4	bengal
1	Boots	4	bengal
2	Cally	4	bengal
4	Eugene	4	bengal
0	Apricot	5	persian
1	Boots	5	persian
2	Cally	5	persian
4	Eugene	5	persian

Inner Join

Only pairs of matching rows appear in the result.



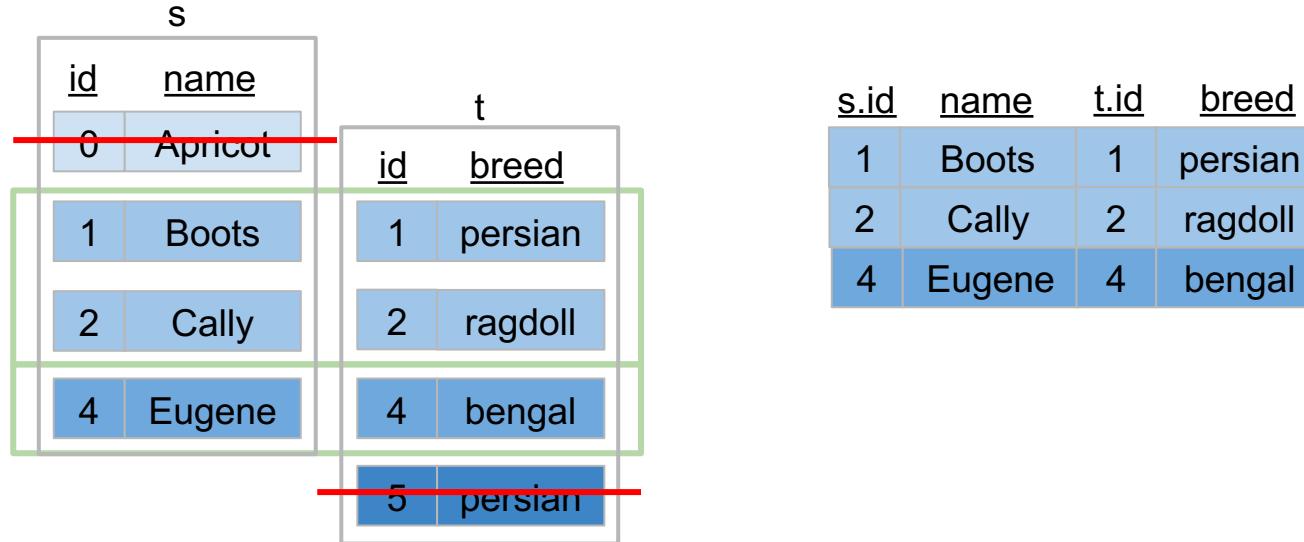
```
SELECT * FROM s JOIN t ON s.id = t.id;
```

```
SELECT * FROM s INNER JOIN t ON s.id = t.id;
```

```
SELECT * FROM s, t WHERE s.id = t.id;
```

Inner Join

Only pairs of matching rows appear in the result.



```
SELECT * FROM s JOIN t ON s.id = t.id;
```

```
SELECT * FROM s INNER JOIN t ON s.id = t.id;
```

```
SELECT * FROM s, t WHERE s.id = t.id;
```

Relationship Between Cross Joins and Inner Join

Conceptually, an inner join is a cross join followed by removal of bad rows.

s		t	
<u>id</u>	<u>name</u>	<u>id</u>	<u>breed</u>
0	Apricot		
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal
		5	persian

<u>s.id</u>	<u>name</u>	<u>t.id</u>	<u>breed</u>
0	Apricot	1	persian
1	Boots	1	persian
2	Cally	1	persian
4	Eugene	1	persian
0	Apricot	2	ragdoll
1	Boots	2	ragdoll
2	Cally	2	ragdoll
4	Eugene	2	ragdoll

(... continued)

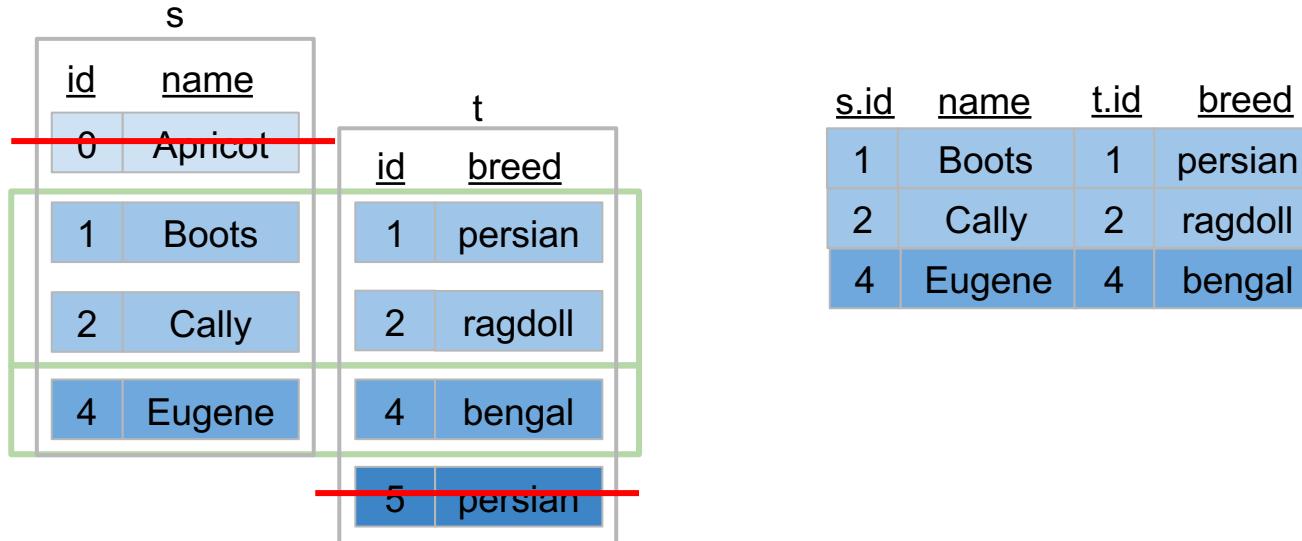
<u>s.id</u>	<u>name</u>	<u>t.id</u>	<u>breed</u>
0	Apricot	4	bengal
1	Boots	4	bengal
2	Cally	4	bengal
4	Eugene	4	bengal
0	Apricot	5	persian
1	Boots	5	persian
2	Cally	5	persian
4	Eugene	5	persian

(to be continued ...)

SELECT * FROM s, t WHERE s.id = t.id;

Relationship Between Cross Joins and Inner Join

Conceptually, an inner join is a cross join followed by removal of bad rows.



```
SELECT * FROM s, t WHERE s.id = t.id;
```

Left Outer Join

Every row in the first table appears in the result, matching or not.

s		t	
<u>id</u>	<u>name</u>	<u>id</u>	<u>breed</u>
0	Apricot	1	persian
1	Boots	2	ragdoll
2	Cally	4	bengal
4	Eugene	5	persian

```
SELECT * FROM s LEFT JOIN t ON s.u = t.v;
```

Left Outer Join

Every row in the first table appears in the result, matching or not.

s		t	
<u>id</u>	<u>name</u>	<u>id</u>	<u>breed</u>
0	Apricot	1	persian
1	Boots	2	ragdoll
2	Cally	4	bengal
4	Eugene	5	persian

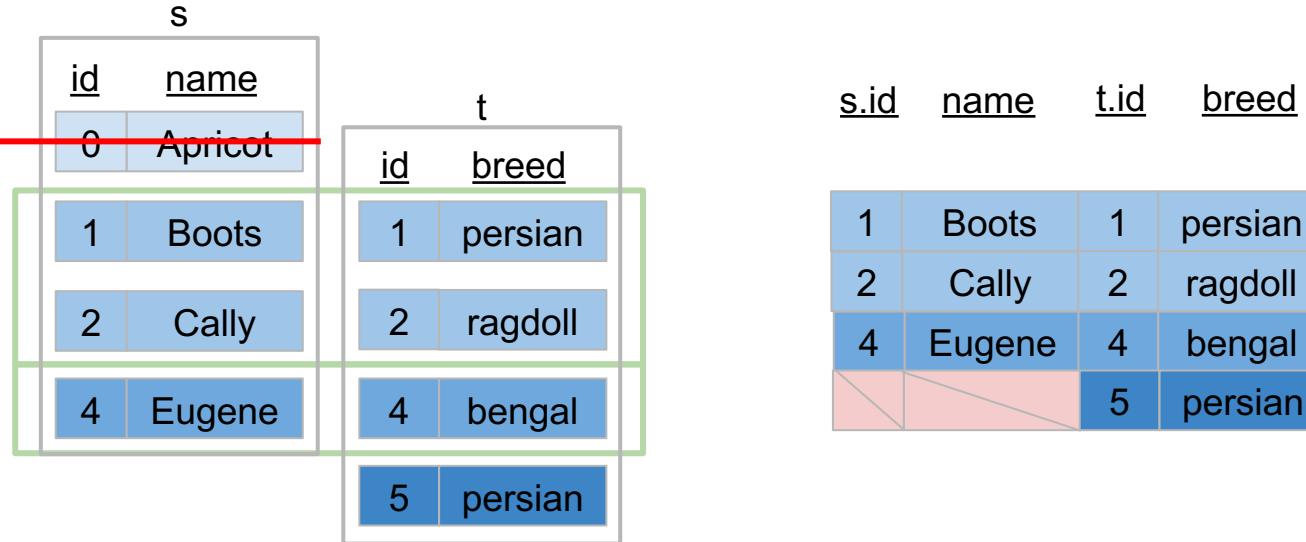
<u>s.id</u>	<u>name</u>	<u>t.id</u>	<u>breed</u>
0	Apricot		
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal

Missing values are null.

```
SELECT * FROM s LEFT JOIN t ON s.u = t.v;
```

Right Outer Join

Every row in the second table appears in the result, matching or not.



```
SELECT * FROM s RIGHT JOIN t ON s.u = t.v;
```

Note: SQLite does not implement
RIGHT JOIN.

Full Outer Join

Every row in both tables appears, matching or not.

s		t	
<u>id</u>	<u>name</u>	<u>id</u>	<u>breed</u>
0	Apricot	1	persian
1	Boots	2	ragdoll
2	Cally	4	bengal
4	Eugene	5	persian

<u>s.id</u>	<u>name</u>	<u>t.id</u>	<u>breed</u>
0	Apricot		
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal
		5	persian

```
SELECT * FROM s FULL OUTER JOIN t ON s.u = t.v;
```

Note: SQLite does not support FULL OUTER JOIN.

Other Join Conditions

We can join on conditions other than equality.

student	
<u>age</u>	<u>name</u>
29	Jameel
37	Jian
20	John
20	Emma

teacher	
<u>age</u>	<u>breed</u>
52	Ira
41	Husain
27	John
36	Anuja

```
SELECT * FROM student, teacher WHERE student.age > teacher.age;
```

Other Join Conditions

We can join on conditions other than equality.

student	
age	name
29	Jameel
37	Jian
20	John
20	Emma

teacher	
age	breed
52	Ira
41	Husain
27	John
36	Anuja

29	Jameel	27	John
37	Jian	27	John
37	Jian	36	Anuja

Note that every satisfying pair appears.

- Inner joins are just cross joins followed by removing rows that don't match.

```
SELECT * FROM student, teacher WHERE student.age > teacher.age;
```

Databases are a more sophisticated way to store data than simple CSV or similar files.

For DS purposes the advantages are:

- Ability to interact with large datasets.
- Ability to harness SQL syntax, which can be simpler in some situations. Examples:
 - Applying multiple aggregation functions.
 - Joins