LECTURE 22

SQL II and Data Serialization

SQL and Databases: An alternative to Pandas and CSV files.
Data serialization: beyond the CSV (in different directions).

Data 100/Data 200, Spring 2023 @ UC Berkeley

Narges Norouzi and Lisa Yan

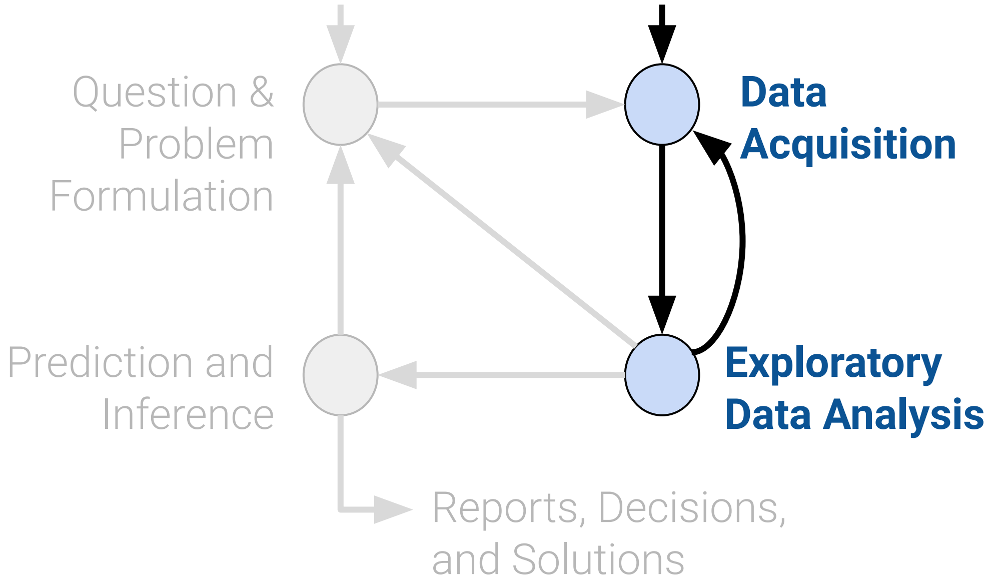
slido

Join at slido.com
#3310273

ⓘ Start presenting to display the joining instructions on this slide.

SQL and Databases: An Alternative to Pandas and CSV Files

3310273



(today)

SQL and Databases I → SQL and Databases II

Summary So Far

SQL Query Structure

```
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>];
```

- By convention, use **all caps** for keywords in SQL statements.
- Use **newlines** to make SQL code more readable.
- **AS** keyword: rename columns during selection process.
- `WHERE`: rows; `HAVING`: groups. **`WHERE` precedes `HAVING`**.
- Column Expressions may include aggregation functions (`MAX`, `MIN`, etc.).

Warmup: HAVING vs. WHERE

What will be the return relation?

```
SELECT type, MAX(name)
FROM DishDietary
WHERE notes == 'gf'
GROUP BY type
HAVING MAX(cost) <= 7;
```

name	type	cost	notes
ravioli	entree	10	dairy
ramen	entree	7	pork
taco	entree	7	gf
edamame	appetizer	4	gf
fries	appetizer	4	gf
potsticker	appetizer	4	pork
ice cream	dessert	5	dairy

DishDietary

3310273

A.

type
appetizer
entree

B.

type	MAX(name)
appetizer	None
entree	None

C.

type	MAX(name)
appetizer	edamame
entree	ramen

D.

type	MAX(name)
appetizer	fries
entree	taco

E.

type	MAX(name)
appetizer	fries
entree	taco
dessert	ice cream

F. Something else

slido

What will be the returned relation? 1

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

Warmup: HAVING vs. WHERE

What will be the return relation

```
SELECT type, MAX(name)
FROM DishDietary
WHERE notes == 'gf'
GROUP BY type
HAVING MAX(cost) <= 7;
```

name	type	cost	notes
ravioli	entree	10	dairy
ramen	entree	7	pork
taco	entree	7	gf
edamame	appetizer	4	gf
fries	appetizer	4	gf
potsticker	appetizer	4	pork
ice cream	dessert	5	dairy

DishDietary

A.

type
appetizer
entree

B.

type	MAX(name)
appetizer	None
entree	None

C.

type	MAX(name)
appetizer	edamame
entree	ramen

D.

type	MAX(name)
appetizer	fries
entree	taco

E.

type	MAX(name)
appetizer	fries
entree	taco
dessert	ice cream

F. Something else

To filter:

- Rows, use **WHERE**.
- Groups, use **HAVING**.

WHERE precedes **HAVING**.

3310273

Order of Execution

3310273

A query is **not** evaluated according to Python operator precedence.

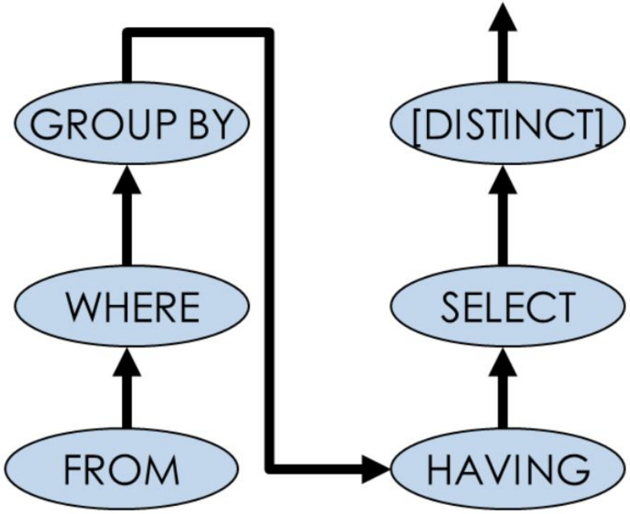
```
SELECT <column expression list>
FROM <table>
[WHERE <predicate>]
[GROUP BY <column list>]
[HAVING <predicate>]
[ORDER BY <column list>]
[LIMIT <number of rows>]
[OFFSET <number of rows>];
```

SQL Query structure

Generally, the order of execution of clauses within a statement are:

1. `FROM`: Retrieve the Relations.
2. `WHERE`: Filter the rows.
3. `GROUP BY`: Make groups.
4. `HAVING`: Filter the groups.
5. `SELECT`: aggregate into rows, get specific columns.
6. `DISTINCT`: ???

Let's check it out!



Query clause precedence

DISTINCT, LIKE, CAST

Lecture 22, Data 100 Spring 2023

3310273

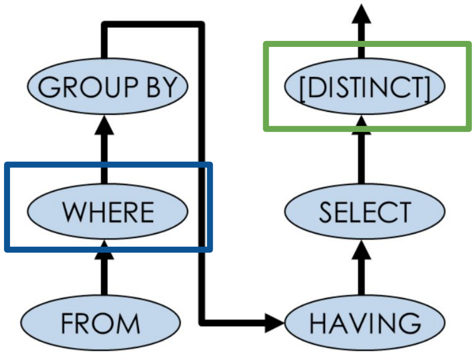
DISTINCT: What does this do?

```
SELECT DISTINCT type
FROM Dish
WHERE cost < 8;
```

type
entree
appetizer
dessert

name	type	cost
ravioli	entree	10
ramen	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish



3310273

DISTINCT: What does this do?

```
SELECT DISTINCT type
FROM Dish
WHERE cost < 8;
```

type
entree
appetizer
dessert

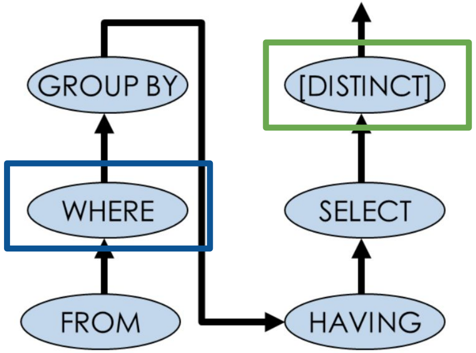
```
SELECT DISTINCT type, cost
FROM Dish
WHERE cost < 11;
```

type	cost
entree	10
entree	7
appetizer	4
dessert	5

name	type	cost
ravioli	entree	10
ramen	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

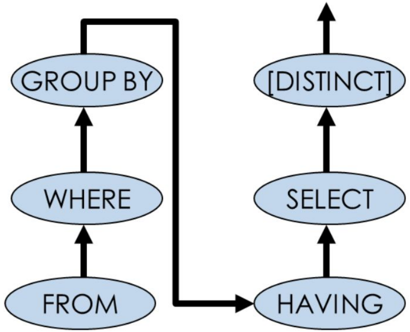
Dish

3310273



DISTINCT creates unique tuples.
WHERE precedes **DISTINCT**.

DISTINCT vs. GROUP BY



3310273

These queries both return unique tuples (ignoring row order) through different precedence of clauses:

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

type	cost
entree	10
entree	7
appetizer	4
dessert	5

name	type	cost
ravioli	entree	10
ramen	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Select only unique tuples.

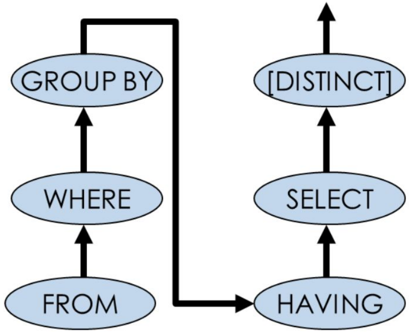
```
SELECT
    type,
    cost
FROM Dish
GROUP BY type, cost;
```

type	cost
appetizer	4
dessert	5
entree	7
entree	10

name	type	cost
ravioli	entree	10
ramen	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

First group by unique keys, then select group keys.

DISTINCT vs. GROUP BY



3310273

These queries both return unique tuples (ignoring row order) through different precedence of clauses:

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

type	cost
entree	10
entree	7
appetizer	4
dessert	5

Better style to use **SELECT DISTINCT** for unique values/tuples.

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

type	cost
appetizer	4
dessert	5
entree	7
entree	10

I think of this as a degenerate use of GROUP BY, because no aggregate functions.

🌟DISTINCT can also be used in Column Expressions!

3310273

Common query: **GROUP BY** and **DISTINCT** <u>**together**</u>.

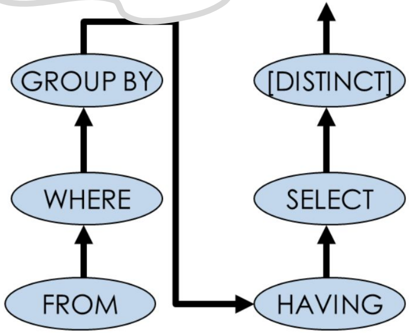
```
SELECT type, AVG(DISTINCT cost)
FROM Dish
GROUP BY type;
```

type	AVG(DISTINCT cost)
appetizer	4.0
dessert	5.0
entree	8.5

Average of the 7 and 10, which are the unique cost values for entrees.

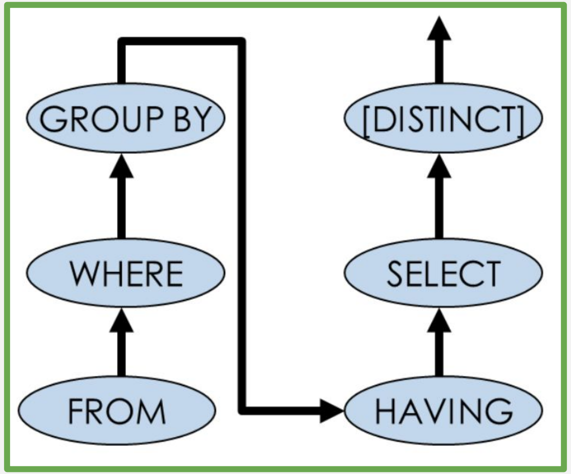
name	type	cost
ravioli	entree	10
ramen	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish



Summary of today (new)

```
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>];
```



- By convention, use **all caps** for keywords in SQL statements.
- Use **newlines** to make SQL code more readable.
- **AS** keyword: rename columns during selection process.
- `WHERE`: rows; `HAVING`: groups. `WHERE` precedes `HAVING`.
- Column Expressions may include aggregation functions (`MAX`, `MIN`, etc.) **and `DISTINCT`.**

Working with TEXT values: LIKE and CAST

3310273

The `LIKE` operator tests whether a string matches a pattern (W3Schools documentation).

- Similar to a regex, but much simpler syntax.
- For example, to select rows where the time string is on the hour, such as 8:00 or 12:00 pm:

```
SELECT *
FROM Timestamps
WHERE time LIKE '%:00%';
```

Two wildcards often used in conjunction with the **`LIKE`** operator:

- `%`: represents zero, one, or multiple characters
- `_`: represents one, single character

Working with TEXT values: LIKE and CAST

3310273

The `LIKE` operator tests whether a string matches a pattern (W3Schools documentation).

- Similar to a regex, but much simpler syntax.
- For example, to select rows where the time string is on the hour, such as 8:00 or 12:00 pm:

```
SELECT *
FROM Timestamp
WHERE time LIKE '%:00%';
```

Two wildcards often used in conjunction with the **`LIKE`** operator:

- `%`: represents zero, one, or multiple characters
- `_`: represents one, single character

The **CAST** keyword converts a table column to another type (W3Schools documentation).

- For example, this converts `runtimeMinutes` and `startYear` to `int` (renaming the latter to `year`).
- Any missing invalid values are replaced by 0.

```
SELECT
    CAST(runtime AS INT)
    CAST(startYear AS INT) AS year
FROM Movie;
```

SQL and Pandas

Lecture 22, Data 100 Spring 2023

%%sql

In the previous lecture, we 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

Now, we'll see how to use the sqlalchemy Python library to allow communication between pandas and SQL databases.

3310273

Example


```
import sqlalchemy
# create a SQL Alchemy connection to the database
engine = sqlalchemy.create_engine("sqlite:///data/lec18_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

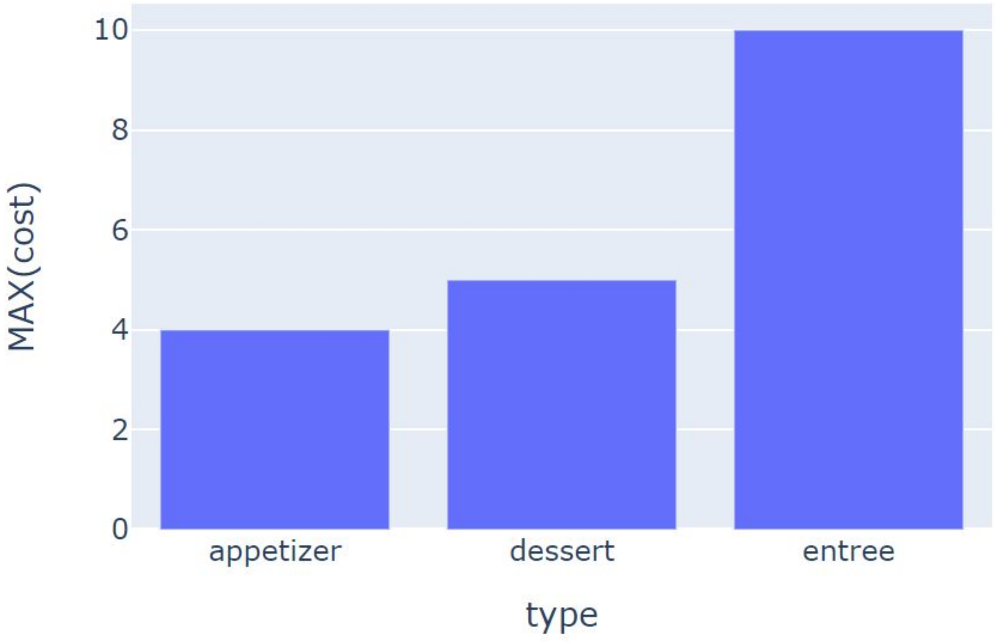
Basic idea:

- 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



%%sql has some tricks up its sleeve!

3310273

Both of these syntaxes work and let you use Python variables in your SQL magics:

```
query = """
SELECT *
FROM Dragon
LIMIT 2
OFFSET 1;
"""
```

```
%%sql
{query}
```

```
 * sqlite:///data/lec18_basic_examples.db
Done.
```

name	year	cute
drogon	2011	-100
dragon 2	2019	0

```
%%sql
$query
```

```
 * sqlite:///data/lec18_basic_examples.db
Done.
```

name	year	cute
drogon	2011	-100
dragon 2	2019	0

In general, use {var} instead of $var, as the former lets you do lots more:

While $var is limited to the value of var, {var} lets you also compute complex expressions. This gives you the full power of Python:

3310273

```
query = """
SELECT * FROM Dish
LIMIT {limit}
"""
```

```
lim = 5
```

```
%%sql res <<
{query.format(limit=lim)}
```

```
 * sqlite:///data/lec18_basic_examples.db
Done.
Returning data to local variable res
```

```
res.sql
```

```
'SELECT * FROM Dish\nLIMIT 5'
```

```
res.DataFrame()
```

	name	type	cost
0	ravioli	entree	10
1	pork bun	entree	7
2	taco	entree	7
3	edamame	appetizer	4
4	fries	appetizer	4

3310273

IMDb Example

Lecture 22, Data 100 Spring 2023

The IMDb Dataset

3310273

IMDb (Internet Movie Database) 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

3310273

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

3310273

To read this file in SQL, we first connect to the database as before.

Updated 4/9

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

We can then request the list of Tables:

```
%%sql tables <<
SELECT sql FROM sqlite_master WHERE type='table';
```

sql

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

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

The IMDB Dataset

3310273

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

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

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

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

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

Getting 10 Rows

3310273

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

```
%%sql
$get_10_movies
```

tconst	titleType	primaryTitle	originalTitle	isAdult	startYear	endYear	runtimeMinutes	genres
tt0000001	short	Carmencita	Carmencita	0	1894	\N	1	Documentary,Short
tt0000002	short	Le clown et ses chiens	Le clown et ses chiens	0	1892	\N	5	Animation,Short
tt0000003	short	Pauvre Pierrot	Pauvre Pierrot	0	1892	\N	4	Animation,Comedy,Romance
tt0000004	short	Un bon bock	Un bon bock	0	1892	\N	12	Animation,Short
tt0000005	short	Blacksmith Scene	Blacksmith Scene	0	1893	\N	1	Comedy,Short
tt0000006	short	Chinese Opium Den	Chinese Opium Den	0	1894	\N	1	Short
tt0000007	short	Corbett and Courtney Before the Kinetograph	Corbett and Courtney Before the Kinetograph	0	1894	\N	1	Short,Sport
tt0000008	short	Edison Kinetoscopic Record of a Sneeze	Edison Kinetoscopic Record of a Sneeze	0	1894	\N	1	Documentary,Short
tt0000009	movie	Miss Jerry	Miss Jerry	0	1894	\N	45	Romance
tt0000010	short	Leaving the Factory	La sortie de l'usine Lumière à Lyon	0	1895	\N	1	Documentary,Short

Finding Action Movies

3310273

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

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

```
%%sql action_movies <<
{action_movies_query}
```

Finding Action Movies

3310273

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

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

```
%%sql action_movies <<
{action_movies_query}
```

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

```
action_movies.DataFrame()
```

	id	title	time	year
0	tt0000574	The Story of the Kelly Gang	70	1906
1	tt0002574	What Happened to Mary	150	1912
2	tt0003545	Who Will Marry Mary?	\N	1913
3	tt0003747	Cameo Kirby	50	1914
4	tt0003897	The Exploits of Elaine	220	1914
...	...	...	...	...
42674	tt9904270	Get Rid of It	\N	\N
42675	tt9904682	SIUAT	\N	\N
42676	tt9905492	Midnight Reckoning	\N	\N
42677	tt9905708	Résilience	\N	\N
42678	tt9907670	Wanderer in a Business Suit	\N	1961

42679 rows × 4 columns

Three Problems With Our Data

3310273

- 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.

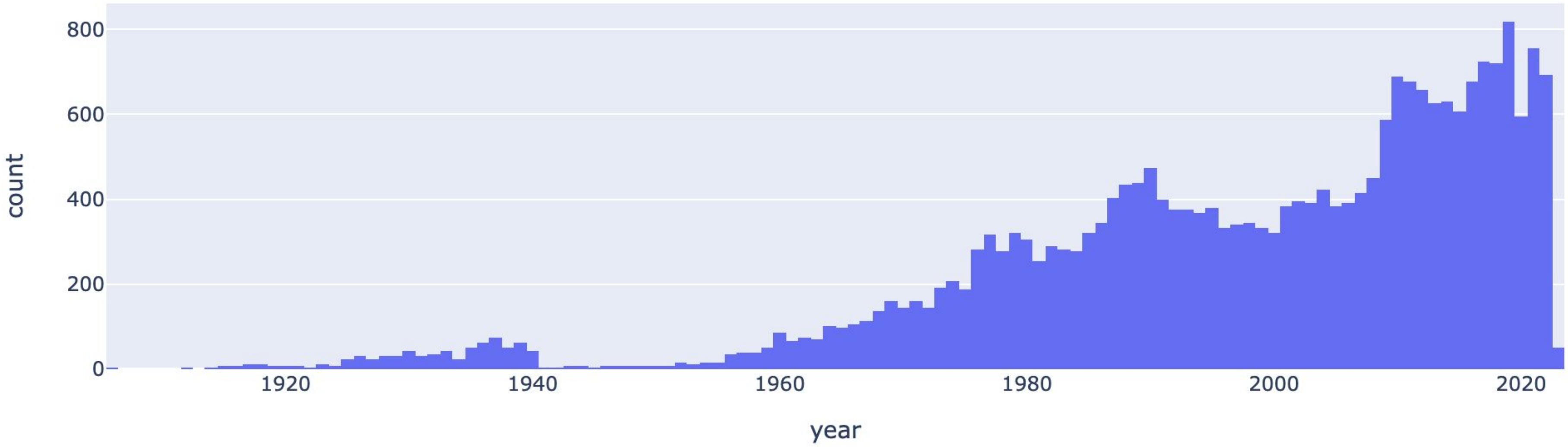
```
CREATE TABLE "titles"(
  "tconst" TEXT,
  "titleType" TEXT,
  "primaryTitle" TEXT,
  "originalTitle" TEXT,
  "isAdult" TEXT,
  "startYear" TEXT,
  "endYear" TEXT,
  "runtimeMinutes" TEXT,
  "genres" TEXT
)
```

	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
...	...	...	...	...

Visualization

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

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



3310273

Result of Our Query

3310273

```
%%sql action_movies <<
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
```

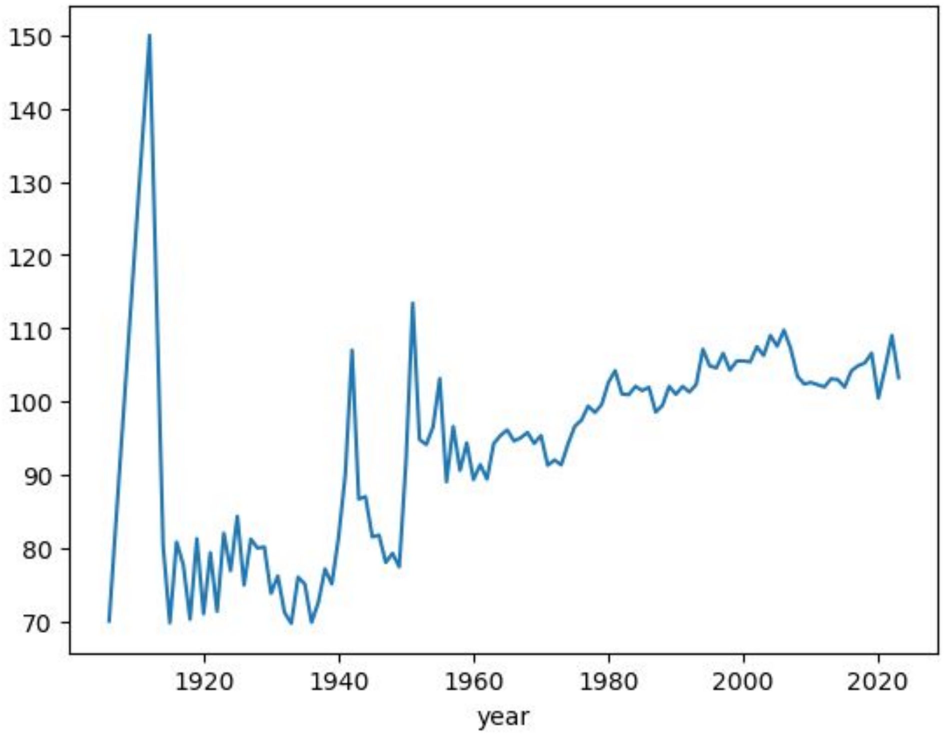
	id	title	time	year
0	tt0000574	The Story of the Kelly Gang	70	1906
1	tt0002574	What Happened to Mary	150	1912
2	tt0004223	The Life of General Villa	105	1914
3	tt0004450	Die Pagode	82	1917
4	tt0004635	The Squaw Man	74	1914
...	...	...	...	...
24385	tt9900748	The Robinsons	110	2019
24386	tt9900782	Kaithi	145	2019
24387	tt9900908	Useless Handcuffs	89	1969
24388	tt9901162	The Robinsons	90	2020
24389	tt9904066	Fox Hunting	66	2019

24390 rows × 4 columns

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();
```

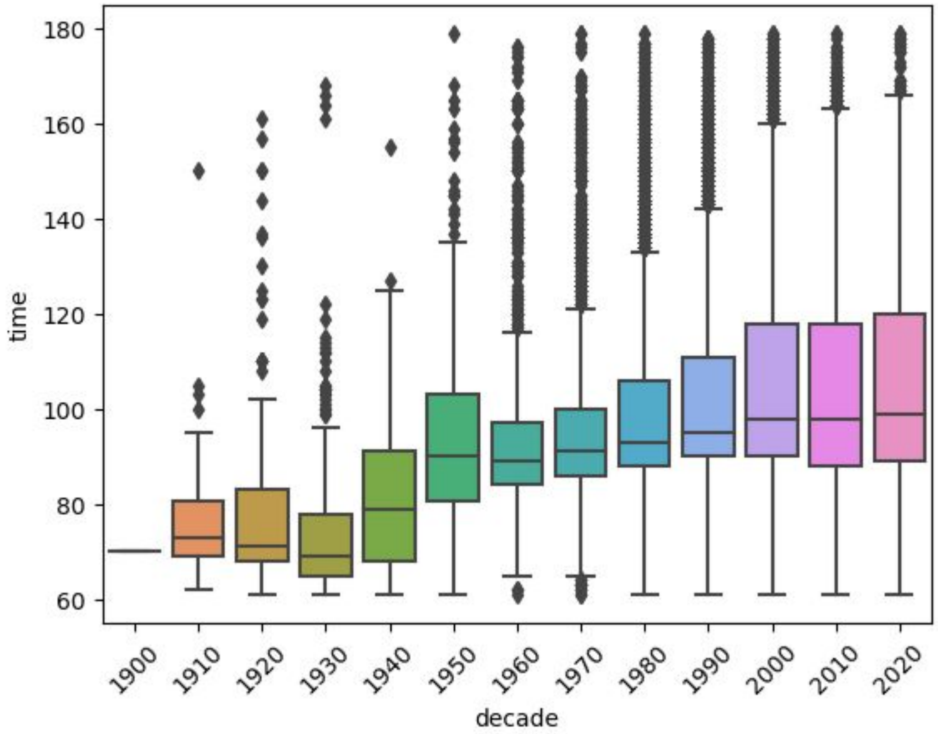


Visualization

3310273

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

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



Interlude

Summer 2023 ASE Applications are out!

Apply to join Data 100 summer course staff!
Priority deadline: **Monday April 10**.
data.berkeley.edu/joining-data-course-staff

EECS/Data Course Staff Side Letter Updates

https://bit.ly/2023-eecs-data-process-video
“Open mic” meeting **tomorrow** (Friday 4/7) 2-3pm, Wheeler 150. Come on by!

General Data Ed Forum

Join! https://edstem.org/us/join/gk5MZQ

SQL Joins

Lecture 22, Data 100 Spring 2023

SQL II

- DISTINCT, LIKE, CAST
- SQL and Pandas
- IMDb Example
- **SQL Joins**

Data Serialization

3310273

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:

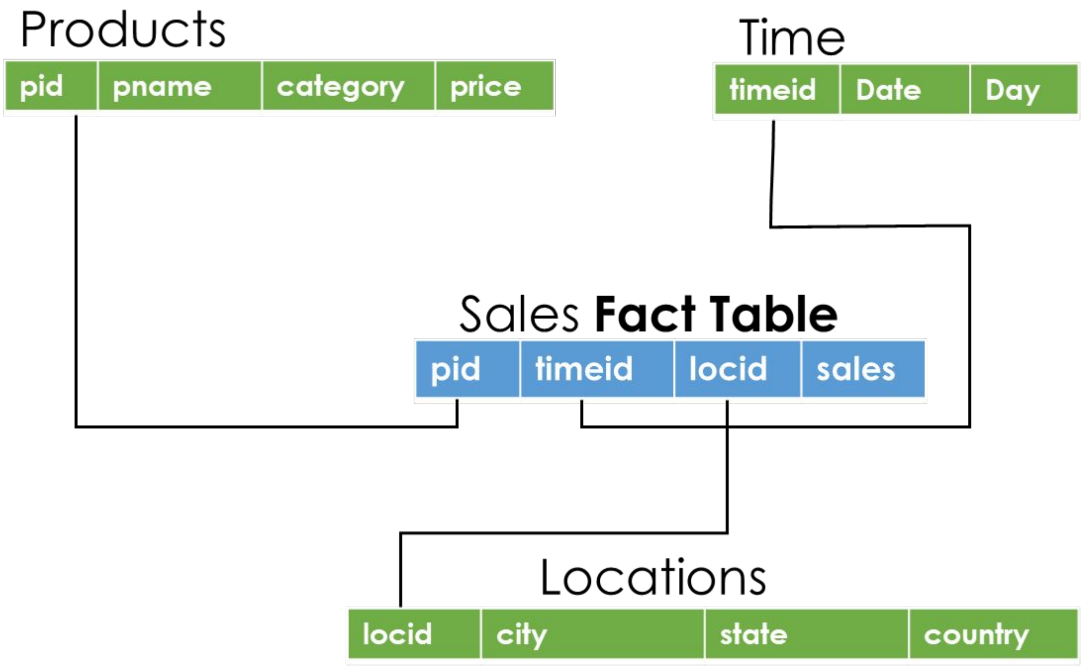
- Easy to manage and summarize.
- Renaming is easy, just change Galaxy 1 to Phablet.

Connections Between Tables

3310273

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

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



Inner Join

3310273

An example of a SQL join is shown below:

latitude	longitude	name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

city	temp
Berkeley	68
Chicago	59
Minneapolis	55

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

name	latitude	temp
Berkeley	122	68
Minneapolis	93	55

This is an "**inner join**". It turns out there are other types of joins.

Cat Breeds

3310273

Persian

Ragdoll

Bengal

Cross Join

3310273

```
SELECT *
FROM s, t;
```

s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

s.id	name	t.id	breed
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...

s.id	name	t.id	breed
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

In a **cross join** (aka Cartesian product), all pairs of rows appear in the result!

slido

What will be the returned relation? 2

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

What if we did an inner join instead?

3310273

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

s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

A.

s.id	name	t.id	breed
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal

B.

id	name	breed
1	Boots	persian
2	Cally	ragdoll
4	Eugene	bengal

C.

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

D. Something else

What if we did an inner join instead?

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

A.

s.id	name	t.id	breed
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal

s

id	name
~~0~~	~~Apricot~~
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
~~5~~	~~persian~~

Only pairs of “matching” rows appear in the result.

Relationship Between Cross Joins and Inner Join

3310273

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

equivalent

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

(inner join is default)

equivalent

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

s

id	name
~~0~~	~~Apricot~~
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
~~5~~	~~persian~~

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

Relationship Between Cross Joins and Inner Join

3310273

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

s

id	name
~~0~~	~~Apricot~~
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
~~5~~	~~persian~~

s.id	name	t.id	breed
~~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...

s.id	name	t.id	breed
~~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~~

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

Other Joins: Left Outer Join

3310273

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

Every row in the **first (left) able** appears in the result, matching or not.

s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

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

Missing values are null.

Other Joins: Right Outer Join

3310273

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

Every row in the **second (right) table** appears in the result, matching or not.

Note: SQLite does not implement `RIGHT JOIN`.

s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

s.id	name	t.id	breed
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal
		5	persian

Other Joins: Full Outer Join


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

Every row in **both tables** appears, matching or not.

Note: SQLite does not support `FULL OUTER JOIN`.

s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

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

Updated diagram post-lecture

3310273

We can join on conditions other than equality.
Pandas can't do this!

```
SELECT *
FROM Student
JOIN Teacher
    ON Student.age > Teacher.age;
```

equivalent

```
SELECT *
FROM Student, Teacher
WHERE Student.age > Teacher.age;
```

Note that every satisfying pair appears!
Again, **inner joins** are just cross joins followed by removing rows that don't match.

Student

age	name
29	Jameel
37	Jianqi
20	John
20	Emma

Teacher

age	name
52	Esme
41	Husain
27	Josh
36	Anuja

Student.age	Student.name	Teacher.age	Teacher.name
29	Jameel	27	Josh
37	Jianqi	27	Josh
37	Jianqi	36	Anuja

Other SQL Keywords

3310273

Covered in discussion/lab/homework:

```
CASE
    WHEN ... THEN …
    ELSE …
    END
```

More SQL resources:

- https://ds100.org/sp23/resources/#sql
- https://www.w3schools.com/sql/sql_ref_case.asp

SQL and Databases Summary

3310273

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

For DS100 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 other than equality joins (also called equi joins).

For more, see Data 101: Data Engineering.

- https://cal-data-eng.github.io/
- Prerequisites:
  - **Data C100** and CS 61B/INFO 206/equivalent.
  - This class will not assume deep experience with databases or big data solutions.

Optional, but useful for implementing personal/grad team projects

Data Serialization

Lecture 22, Data 100 Spring 2023

LECTURE 22

SQL II and Data Serialization

Content credit: Acknowledgments