LECTURE 17

SQL II

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

Agenda

- Filtering Groups
- EDA in SQL
- Joins
- IMDB Demo

Summary so far

SQL Query Structure

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

Filtering Groups

- **Filtering Groups**
- EDA in SQL
- Joins
- IMDB Demo

Filtering Groups With `HAVING`

What if we only want to keep groups that obey a certain condition?

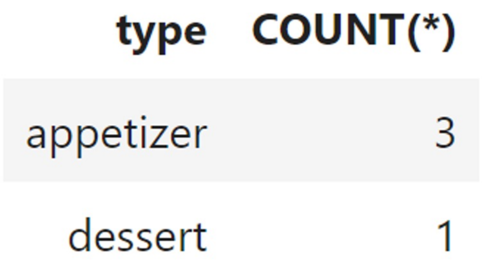
`HAVING` filters groups by applying some condition *across all rows* in each group.

How to interpret: “keep only the groups `HAVING` some condition”

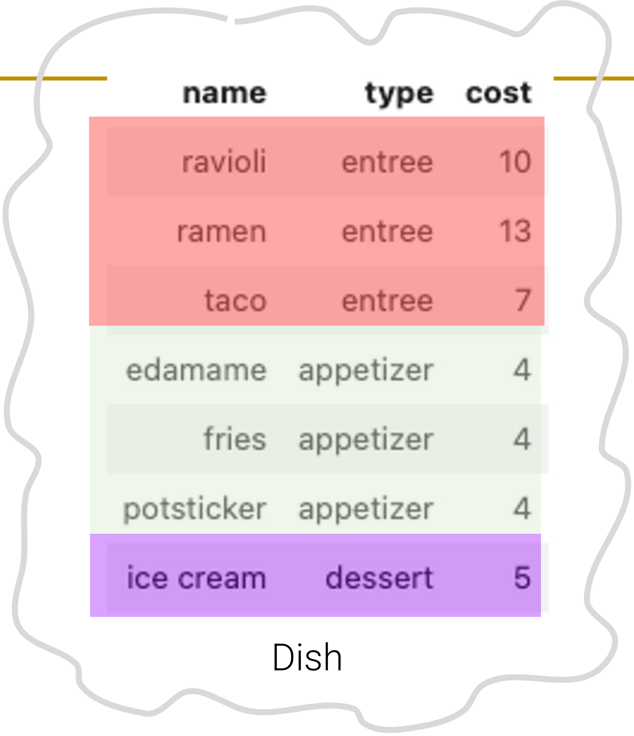
```
SELECT columns
FROM table
GROUP BY grouping_column
HAVING condition_applied_across_group;
```

Filtering Groups With HAVING

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



type	COUNT(*)
appetizer	3
dessert	1



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

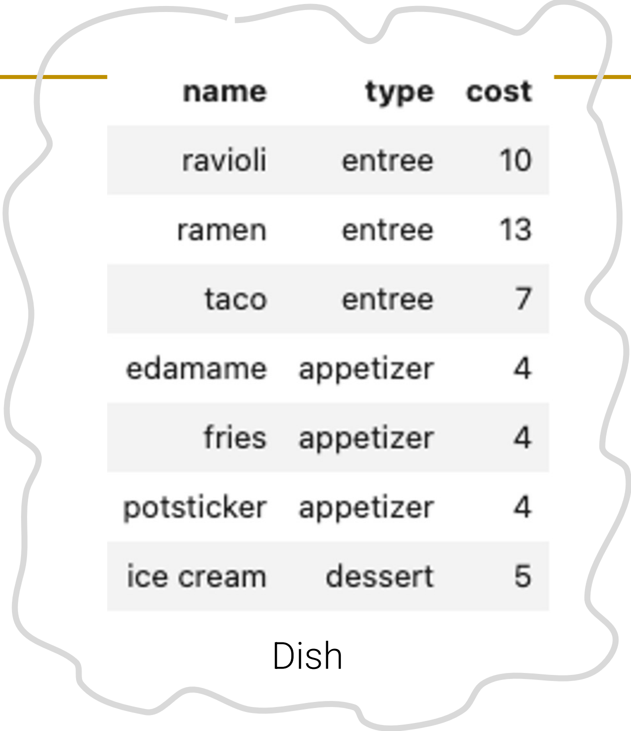
Dish

similar to `groupby("type")`
```
          .filter(lambda f: max(f["cost"]) < 8)
```

WHERE vs. HAVING

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

What will happen here?



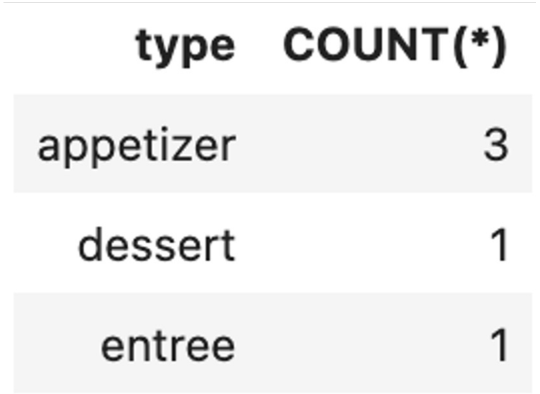
name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

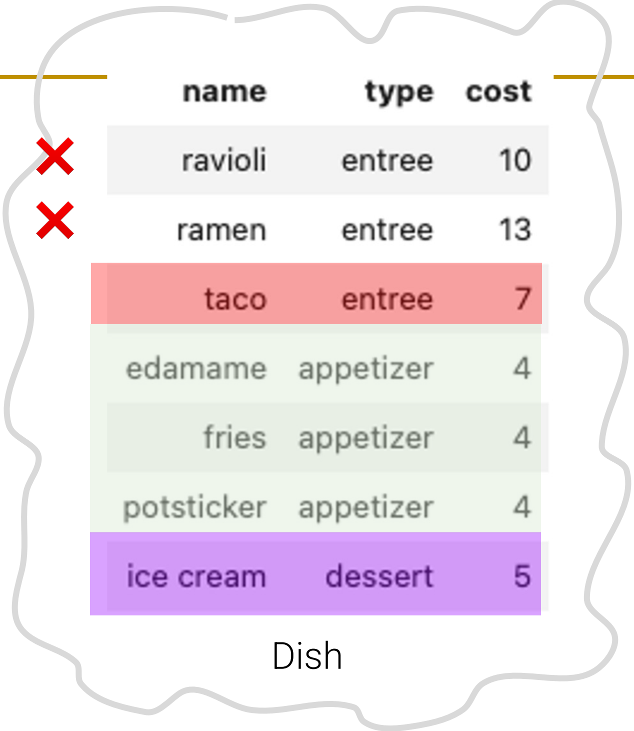
(no slido)

WHERE vs. HAVING

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



type	COUNT(*)
appetizer	3
dessert	1
entree	1



	name	type	cost
✗	ravioli	entree	10
✗	ramen	entree	13
	taco	entree	7
	edamame	appetizer	4
	fries	appetizer	4
	potsticker	appetizer	4
	ice cream	dessert	5

Dish

Animation: WHERE vs. HAVING

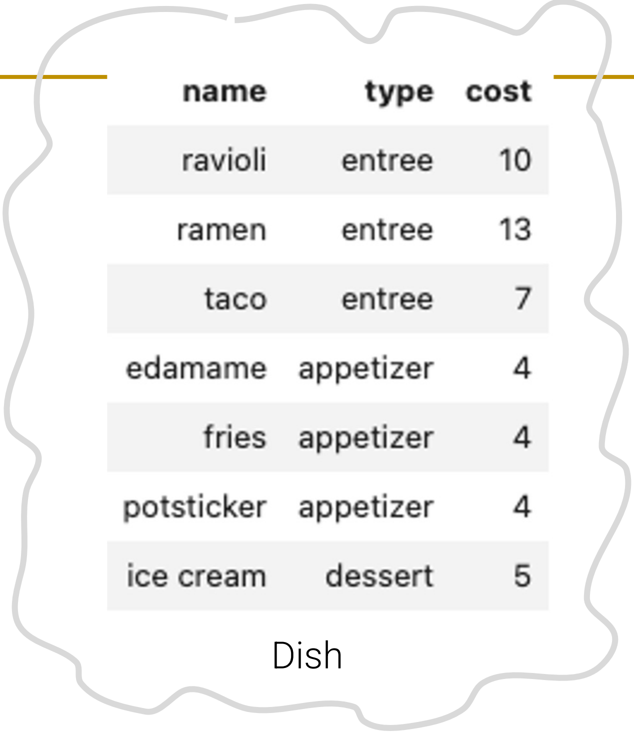
To filter:

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

WHERE precedes **HAVING**.

```
SELECT *
FROM Dish
WHERE cost > 4
GROUP BY type
HAVING MAX(cost) < 10;
```

```
SELECT …
WHERE …
GROUP BY …
HAVING …
```



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

Animation: WHERE vs. HAVING

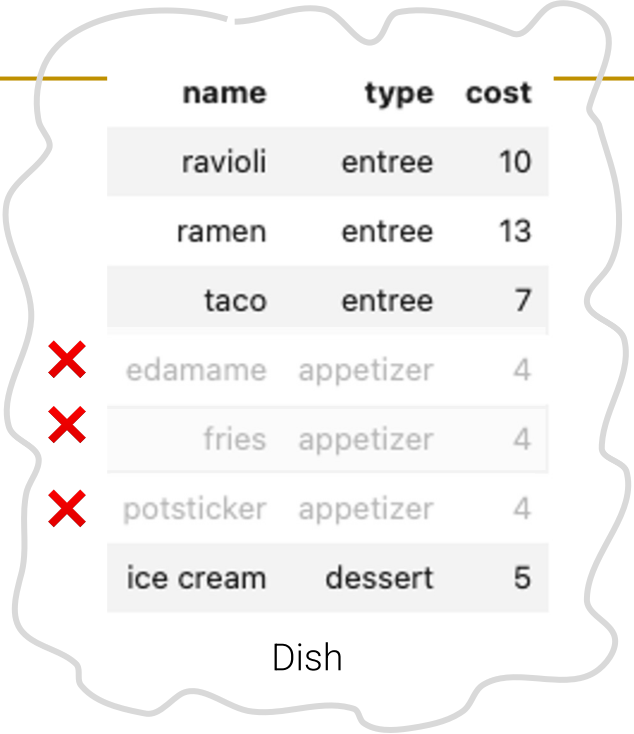
To filter:

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

WHERE precedes **HAVING**.

```
SELECT *
FROM Dish
WHERE cost > 4
GROUP BY type
HAVING MAX(cost) < 10;
```

```
  SELECT …
➡ WHERE …
  GROUP BY …
  HAVING …
```



	name	type	cost
	ravioli	entree	10
	ramen	entree	13
	taco	entree	7
✗	edamame	appetizer	4
✗	fries	appetizer	4
✗	potsticker	appetizer	4
	ice cream	dessert	5

Dish

Animation: WHERE vs. HAVING

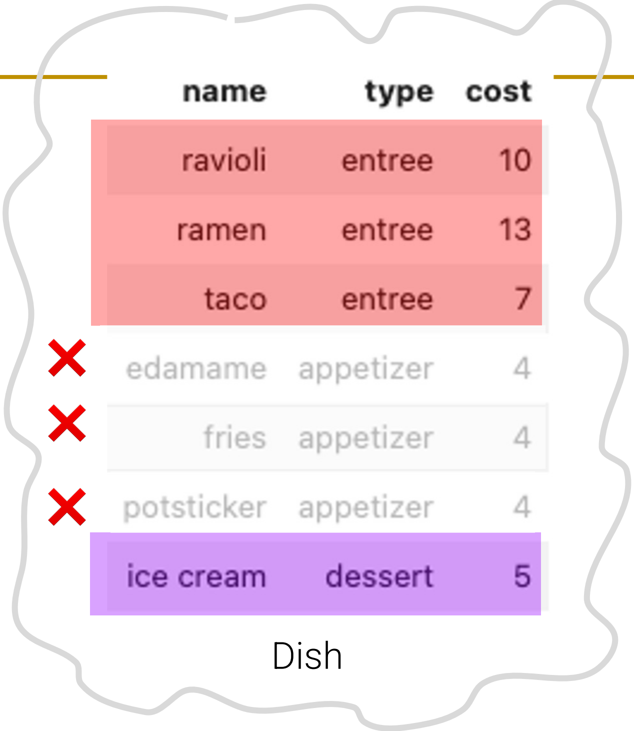
To filter:

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

WHERE precedes **HAVING**.

```
SELECT *
FROM Dish
WHERE cost > 4
GROUP BY type
HAVING MAX(cost) < 10;
```

```
SELECT …
WHERE …
GROUP BY …
HAVING …
```



	name	type	cost
	ravioli	entree	10
	ramen	entree	13
	taco	entree	7
✗	edamame	appetizer	4
✗	fries	appetizer	4
✗	potsticker	appetizer	4
	ice cream	dessert	5

Dish

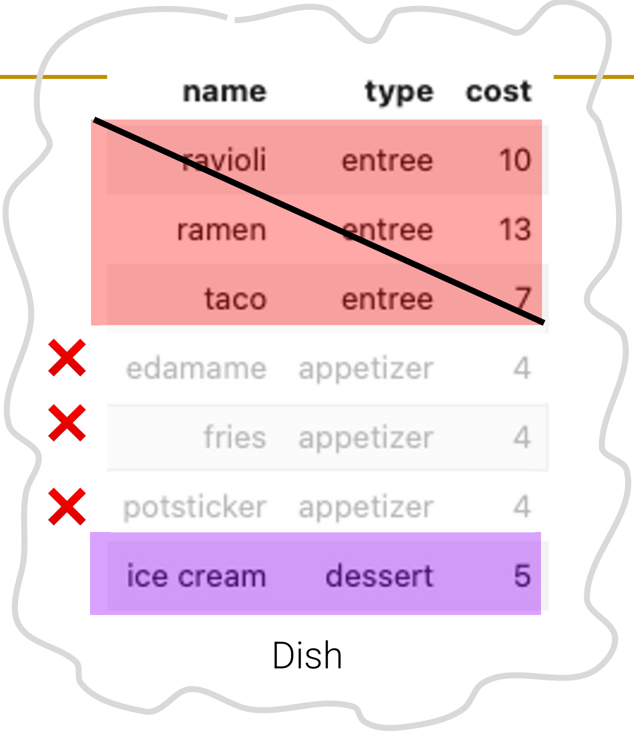
Animation: WHERE vs. HAVING

To filter:

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

WHERE precedes **HAVING**.

```
SELECT *
FROM Dish
WHERE cost > 4
GROUP BY type
HAVING MAX(cost) < 10;
```



	name	type	cost
	~~ravioli~~	~~entree~~	~~10~~
	~~ramen~~	~~entree~~	~~13~~
	~~taco~~	~~entree~~	~~7~~
✗	edamame	appetizer	4
✗	fries	appetizer	4
✗	potsticker	appetizer	4
	ice cream	dessert	5

Dish

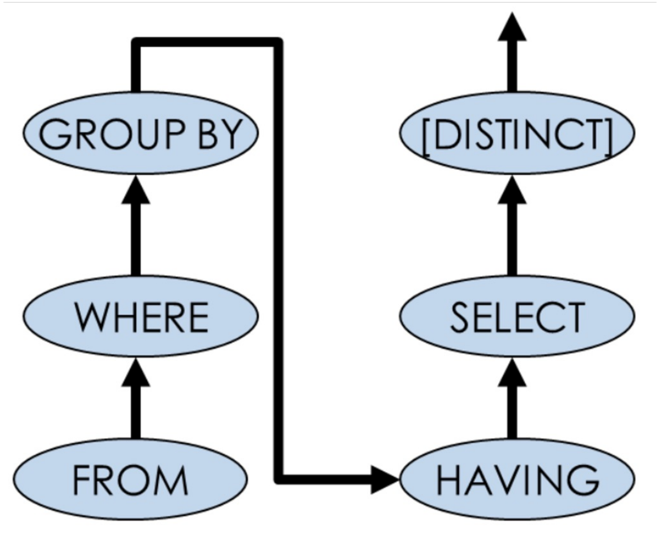
```
SELECT …
WHERE …
GROUP BY …
HAVING …
```

Order of Execution

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

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`: enforce that results must be unique



Summary so far

New keywords

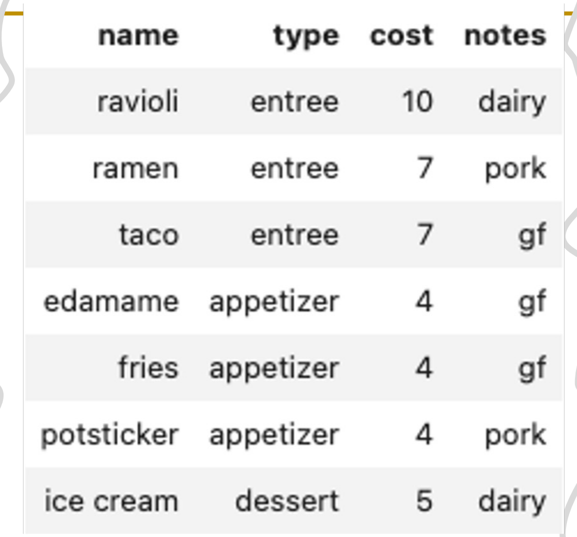
```
SELECT <column expression list>
FROM <table>
[WHERE <predicate>]
[GROUP BY <column list>]
[HAVING <predicate>]
[ORDER BY <column list>]
[LIMIT <number of rows>]
[OFFSET <number of rows>];
```

- 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.**

Quick Check: WHERE vs. HAVING

What will be the return relation?

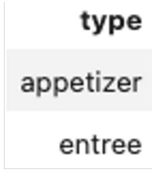
```
SELECT type, MAX(name)
FROM DishDietary
WHERE notes == 'gf'
GROUP BY type
HAVING MAX(cost) <= 7;
```



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

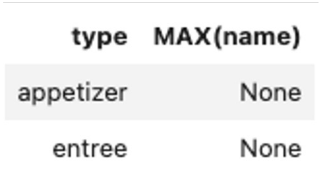
DishDietary

A.



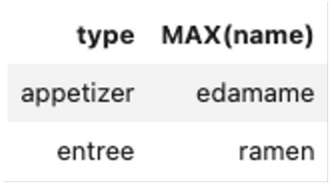
type
appetizer
entree

B.



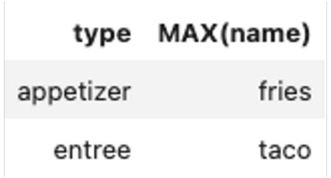
type	MAX(name)
appetizer	None
entree	None

C.



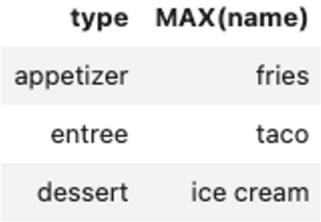
type	MAX(name)
appetizer	edamame
entree	ramen

D.



type	MAX(name)
appetizer	fries
entree	taco

E.



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

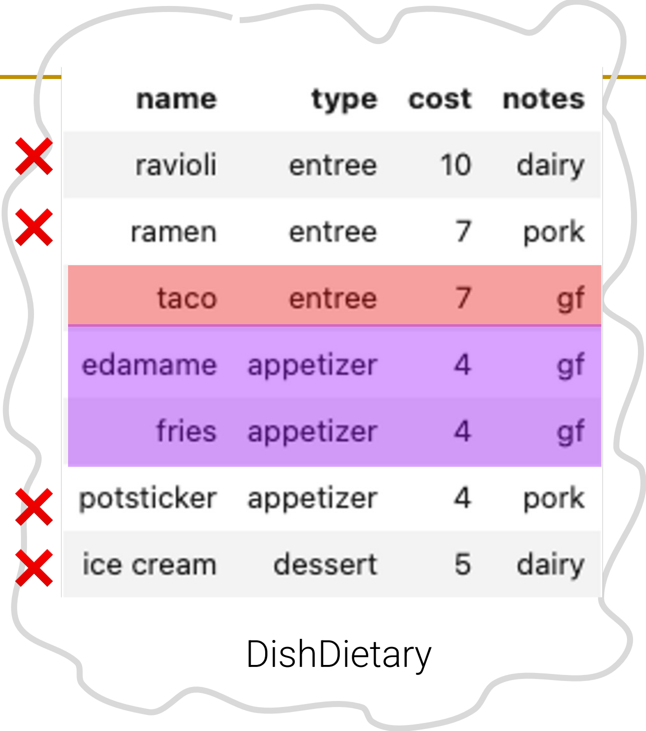
F.

So
mething else

Quick Check: WHERE vs. HAVING

What will be the return relation?

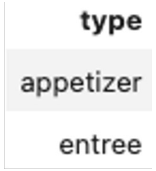
```
SELECT type, MAX(name)
FROM DishDietary
WHERE notes == 'gf'
GROUP BY type
HAVING MAX(cost) <= 7;
```



	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

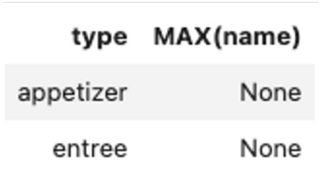
DishDietary

A.



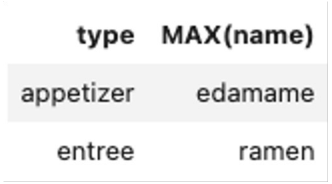
type
appetizer
entree

B.



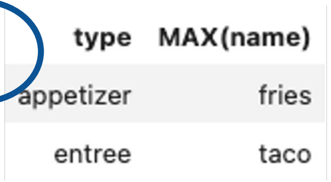
type	MAX(name)
appetizer	None
entree	None

C.



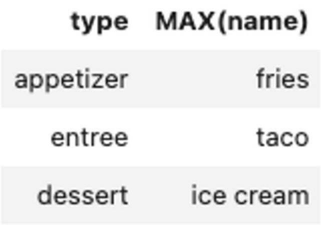
type	MAX(name)
appetizer	edamame
entree	ramen

[✓ D.]



type	MAX(name)
appetizer	fries
entree	taco

E.



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

F.

Something else

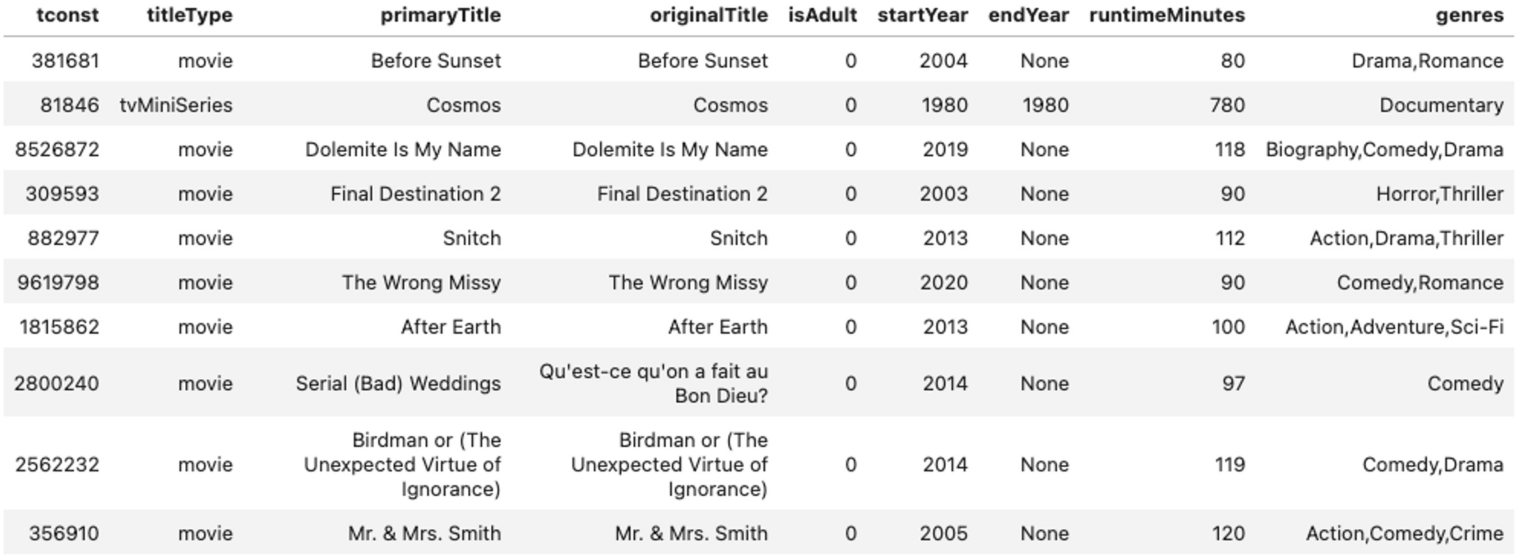
EDA in SQL

- Filtering Groups
- **EDA in SQL**
- Joins
- IMDB Demo

The IMDB Dataset

IMDB = “Internet Movie Database”

Contains information about movies and actors. For example, the `Title` table:



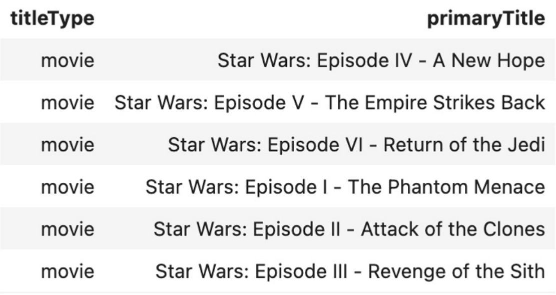
	tconst	titleType	primaryTitle	originalTitle	isAdult	startYear	endYear	runtimeMinutes	genres
	381681	movie	Before Sunset	Before Sunset	0	2004	None	80	Drama,Romance
	81846	tvMiniSeries	Cosmos	Cosmos	0	1980	1980	780	Documentary
	8526872	movie	Dolemite Is My Name	Dolemite Is My Name	0	2019	None	118	Biography,Comedy,Drama
	309593	movie	Final Destination 2	Final Destination 2	0	2003	None	90	Horror,Thriller
	882977	movie	Snitch	Snitch	0	2013	None	112	Action,Drama,Thriller
	9619798	movie	The Wrong Missy	The Wrong Missy	0	2020	None	90	Comedy,Romance
	1815862	movie	After Earth	After Earth	0	2013	None	100	Action,Adventure,Sci-Fi
	2800240	movie	Serial (Bad) Weddings	Qu'est-ce qu'on a fait au Bon Dieu?	0	2014	None	97	Comedy
	2562232	movie	Birdman or (The Unexpected Virtue of Ignorance)	Birdman or (The Unexpected Virtue of Ignorance)	0	2014	None	119	Comedy,Drama
	356910	movie	Mr. & Mrs. Smith	Mr. & Mrs. Smith	0	2005	None	120	Action,Comedy,Crime

Working with Text: `LIKE`

We can perform simple text comparisons in SQL using the **`LIKE`** keyword

How to interpret: “look for entries that are **`LIKE`** the provided example string”

```
SELECT titleType, primaryTitle
FROM Title
WHERE primaryTitle LIKE “%Star Wars%”;
```



titleType	primaryTitle
movie	Star Wars: Episode IV - A New Hope
movie	Star Wars: Episode V - The Empire Strikes Back
movie	Star Wars: Episode VI - Return of the Jedi
movie	Star Wars: Episode I - The Phantom Menace
movie	Star Wars: Episode II - Attack of the Clones
movie	Star Wars: Episode III - Revenge of the Sith

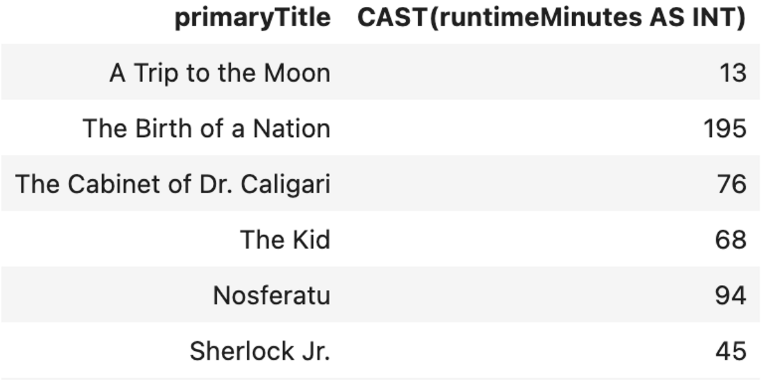
Two “wildcard” characters:

- % means “look for any character, any number of times”
- _ means “look for exactly 1 character”

Converting Data Types: CAST

To convert a column to a different data type, use the CAST keyword as part of the SELECT statement. Returns a *column* of the new data type, which we then SELECT for our output.

```
SELECT primaryTitle, CAST(runtimeMinutes AS INT)
FROM Title;
```



primaryTitle	CAST(runtimeMinutes AS INT)
A Trip to the Moon	13
The Birth of a Nation	195
The Cabinet of Dr. Caligari	76
The Kid	68
Nosferatu	94
Sherlock Jr.	45

Creates a copy of the column with all values of converted to the new data type. We then SELECT this column to include it in the output.

Similar to `.astype` in `pandas`

Applying Conditions: CASE

We create conditional statements (like a Python **if**) using **CASE**

```
CASE WHEN <condition> THEN <value>
     WHEN <other condition> THEN <other value>
     …
     ELSE <yet another value>
     END
```

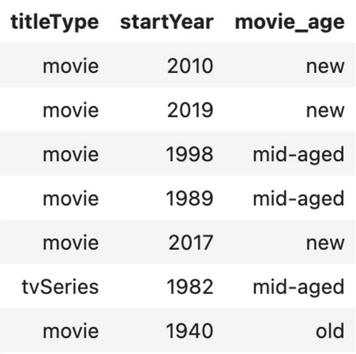
Conceptually, very similar to **CAST** – the **CASE** statement creates a new column, which we then **SELECT** to appear in the output.

Applying Conditions: CASE

We create conditional statements (like a Python `if`) using `CASE`

```
SELECT titleType, startYear,
CASE WHEN startYear < 1950 THEN "old"
     WHEN startYear < 2000 THEN "mid-aged"
     ELSE "new"
     END AS movie_age
FROM Title;
```

All of this occurs within the SELECT statement



titleType	startYear	movie_age
movie	2010	new
movie	2019	new
movie	1998	mid-aged
movie	1989	mid-aged
movie	2017	new
tvSeries	1982	mid-aged
movie	1940	old

Joins

- Filtering Groups
- EDA in SQL
- **Joins**
- IMDB Demo

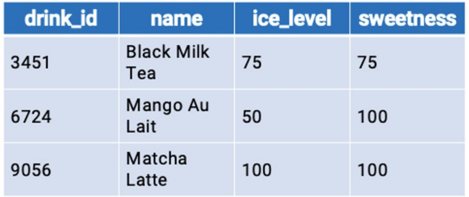
Multidimensional Data

To minimize redundant information, databases typically store data across **fact** and **dimension tables**

Fact table: central table, contains raw facts that typically have pure numerical values. It has information to link its entries to records in other dimension tables. Tends to have few columns, many records.

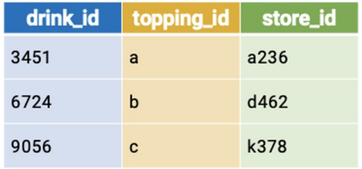
Dimension table: contains more detailed information about each type of fact stored in the fact table (each column). Tends to have more columns and fewer records than fact tables.

Drinks | Dimension Table



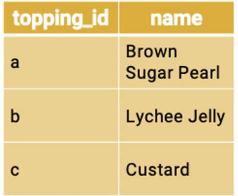
drink_id	name	ice_level	sweetness
3451	Black Milk Tea	75	75
6724	Mango Au Lait	50	100
9056	Matcha Latte	100	100

Products | Fact Table



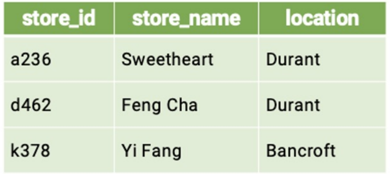
drink_id	topping_id	store_id
3451	a	a236
6724	b	d462
9056	c	k378

Toppings | Dimension Table



topping_id	name
a	Brown Sugar Pearl
b	Lychee Jelly
c	Custard

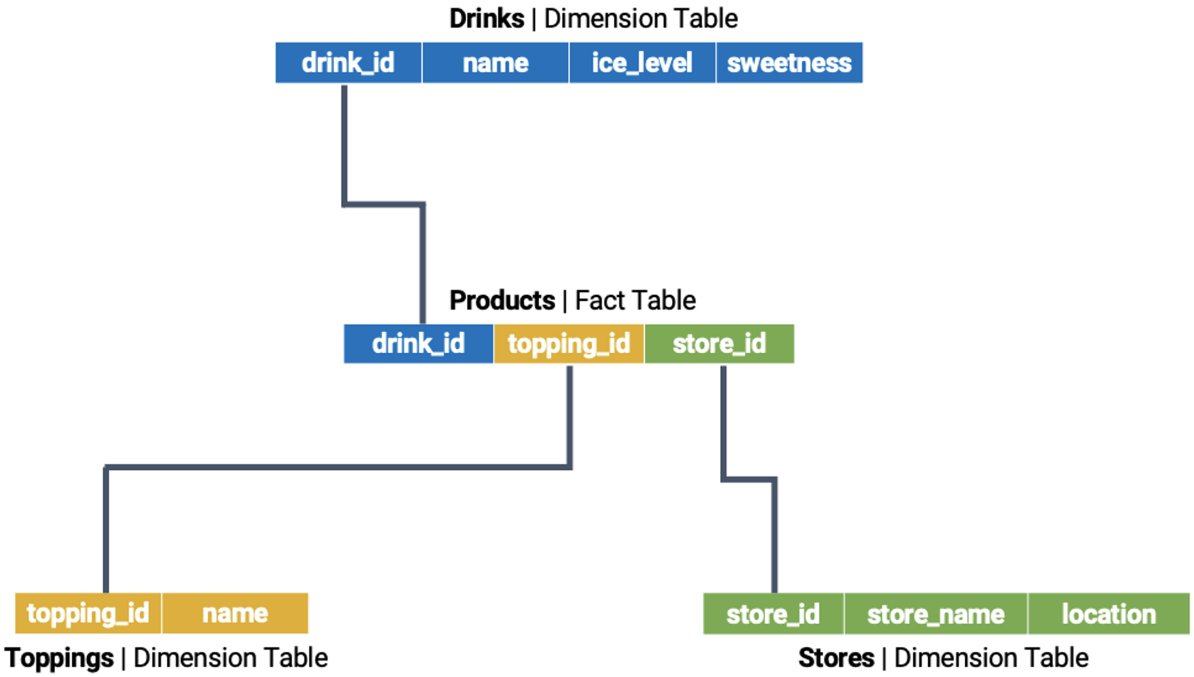
Stores | Dimension Table



store_id	store_name	location
a236	Sweetheart	Durant
d462	Feng Cha	Durant
k378	Yi Fang	Bancroft

Multidimensional Data

A structure that uses fact and dimension tables is called a **star schema**



Drinks | Dimension Table

drink_id | name | ice_level | sweetness

Products | Fact Table

drink_id | topping_id | store_id

topping_id | name

Toppings | Dimension Table

store_id | store_name | location

Stores | Dimension Table

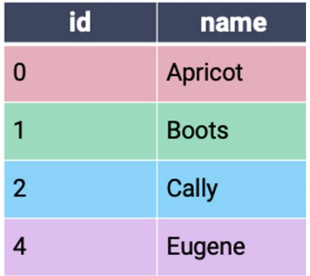
Cats

Persian

Ragdoll

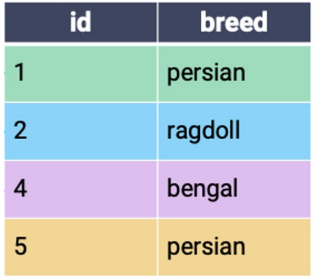
Bengal

s

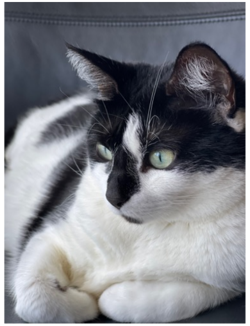


id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t



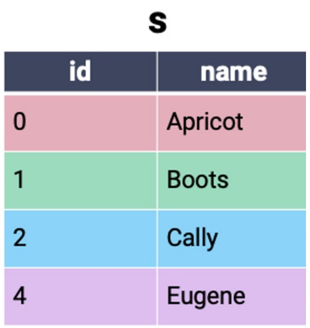
id	breed
1	persian
2	ragdoll
4	bengal
5	persian



Pishi*

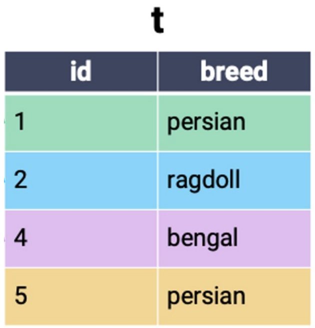
Inner Join

In an **inner join**, we combine every row from the first table with its matching entry in the second table. If a row in one table does not have a match, it is omitted



s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene



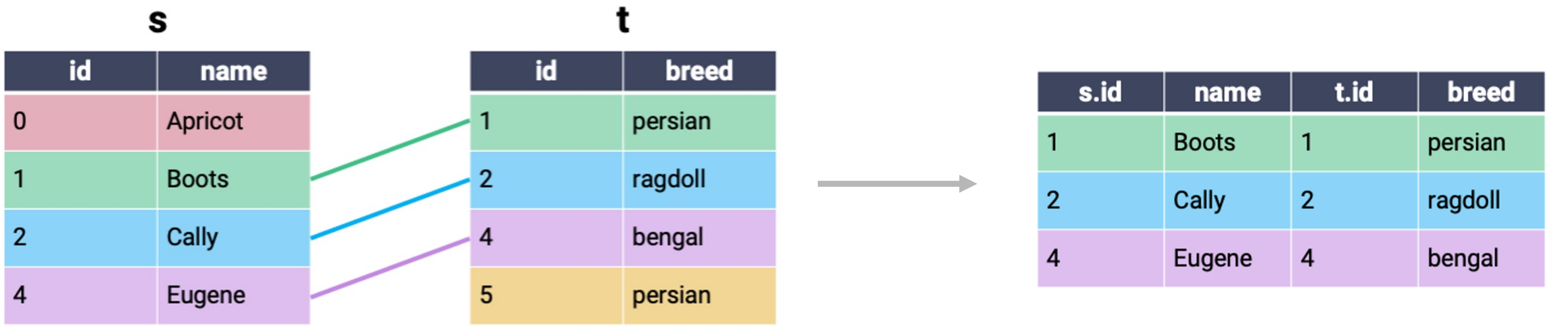
t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

Match rows with the same ID across the tables. Exclude rows with no matching ID

Inner Join

In an **inner join**, we combine every row from the first table with its matching entry in the second table. If a row in one table does not have a match, it is omitted



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

This is the default behavior of `pd.merge`

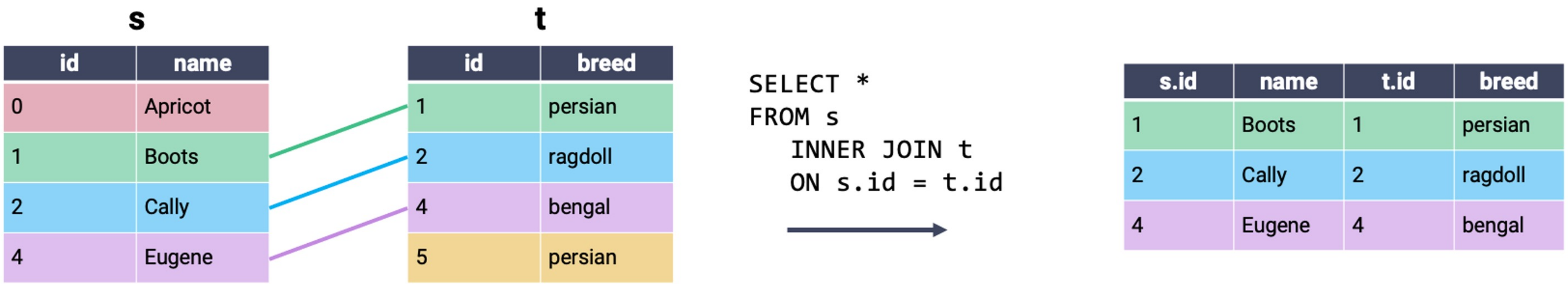
JOIN Syntax

Specify joins between tables as part of the FROM statement

```
SELECT *
FROM table1 INNER JOIN table 2
                 ON table1.key = table2.key
```

Desired type of join

What columns to use to
determine matching entries



s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

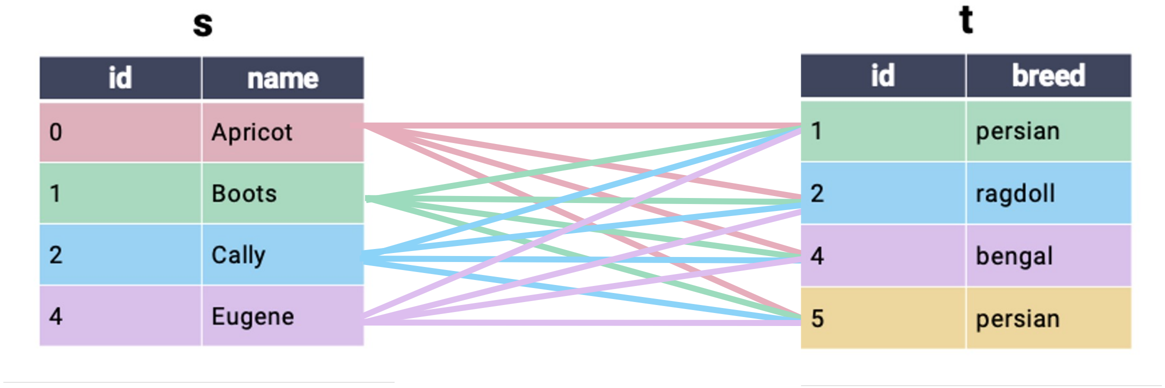
id	breed
1	persian
2	ragdoll
4	bengal
5	persian

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

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

Cross Join

In a **cross join**, we find *every* possible combination of rows across the two tables. A cross join is also called a cartesian product.



s

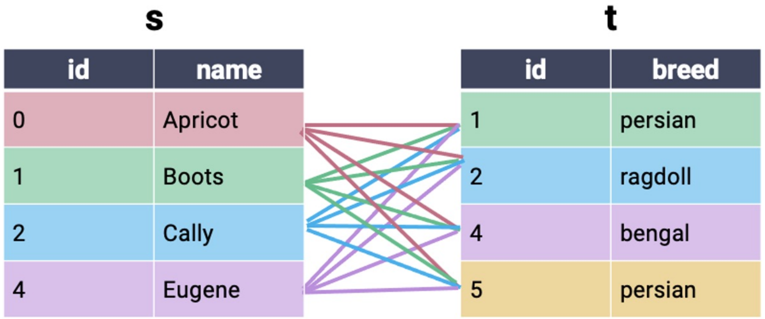
id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

Cross Join

In a **cross join**, we find *every* possible combination of rows across the two tables. A cross join is also called a cartesian product.



s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

```
SELECT *
FROM s
   CROSS JOIN t
```

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

Notice that there is no need to specify a matching key (what columns to use for merging)

Inner Join: Cross Join With Filtering

Conceptually, you can imagine an inner join as a cross join filtered to include only matching rows.

```
SELECT *
FROM s CROSS JOIN t
WHERE s.id = t.id;
```

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

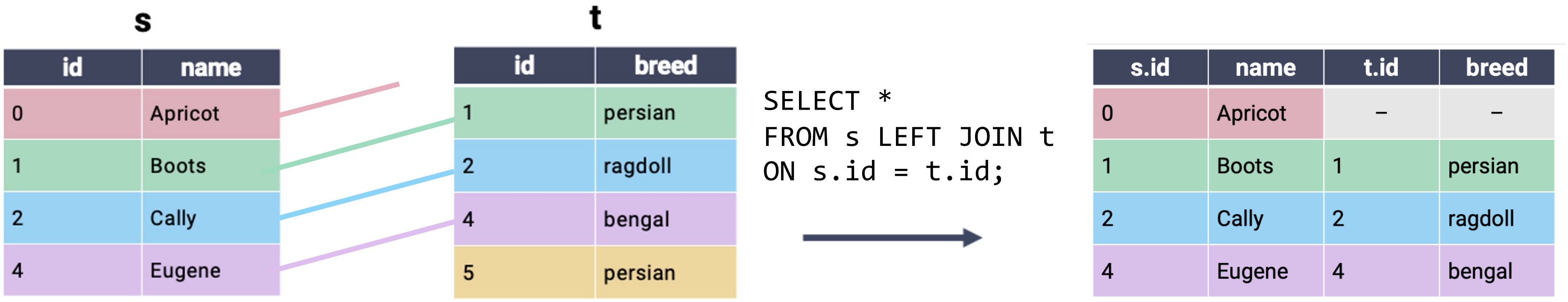
Equivalent

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

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

Left Outer Join

In a **left outer join** (or just **left join**), keep all rows from the left table and *only matching* rows from the right table. Fill NULL for any missing values.



s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

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

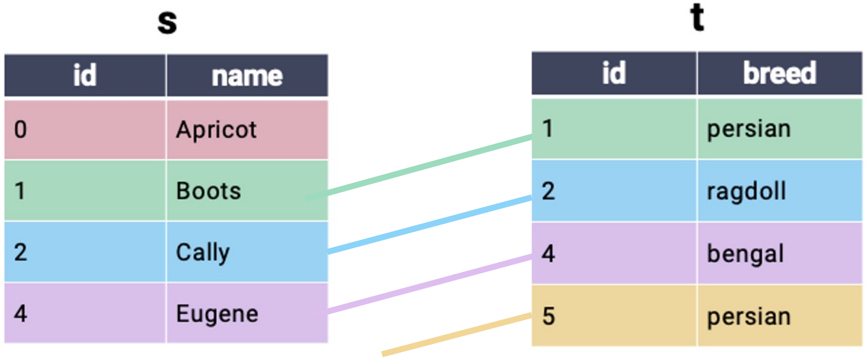
s.id	name	t.id	breed
0	Apricot	–	–
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal

The “left table” is whichever table is referenced first in the JOIN statement.

Fill values without matching entries in the right table with NULL

Right Outer Join

In a **right outer join** (or just **right join**), keep all rows from the right table and *only matching* rows from the right table. Fill NULL for any missing values.



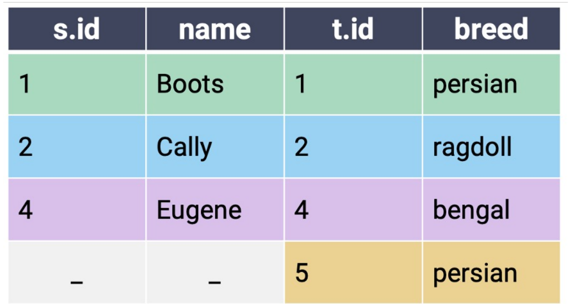
s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

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

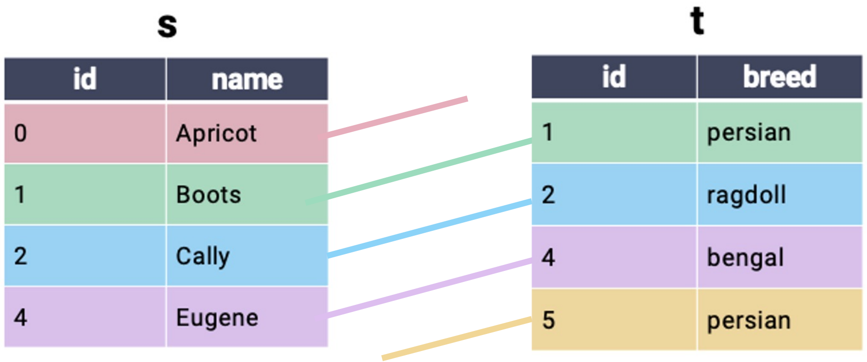


s.id	name	t.id	breed
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal
_	_	5	persian

The "right table" is whichever table is referenced second in the JOIN statement.

Full Outer Join

In a **full outer join**, keep *all rows* from both the left and right tables. Pair any matching rows, then fill missing values with NULL. Conceptually similar to performing both left and right joins.



s

id	name
0	Apricot
1	Boots
2	Cally
4	Eugene

t

id	breed
1	persian
2	ragdoll
4	bengal
5	persian

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

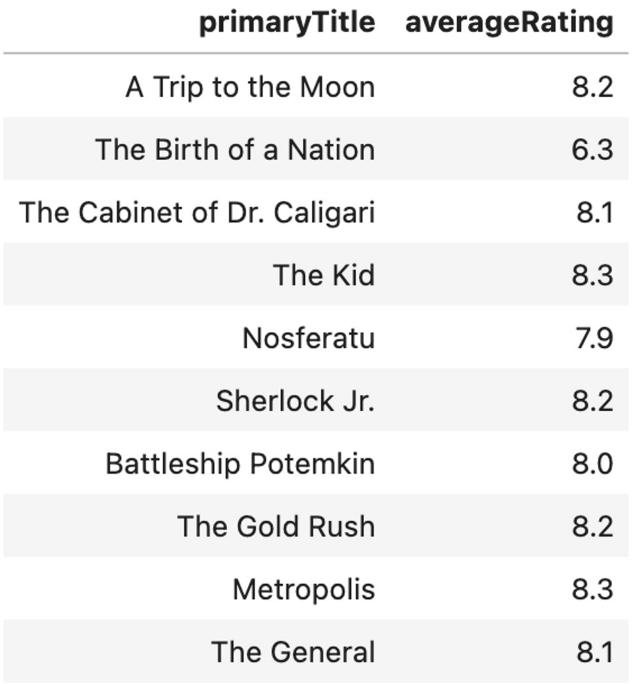
s.id	name	t.id	breed
0	Apricot	–	–
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal
_	_	5	persian

Aliasing in Joins

When working with long table names, we often create aliases that are easier to refer to (just as we did with columns yesterday).

```
SELECT primaryTitle, averageRating
FROM Title AS T INNER JOIN Rating AS R
ON T.tconst = R.tconst;
```

We can then reference columns using the aliased table names



primaryTitle	averageRating
A Trip to the Moon	8.2
The Birth of a Nation	6.3
The Cabinet of Dr. Caligari	8.1
The Kid	8.3
Nosferatu	7.9
Sherlock Jr.	8.2
Battleship Potemkin	8.0
The Gold Rush	8.2
Metropolis	8.3
The General	8.1

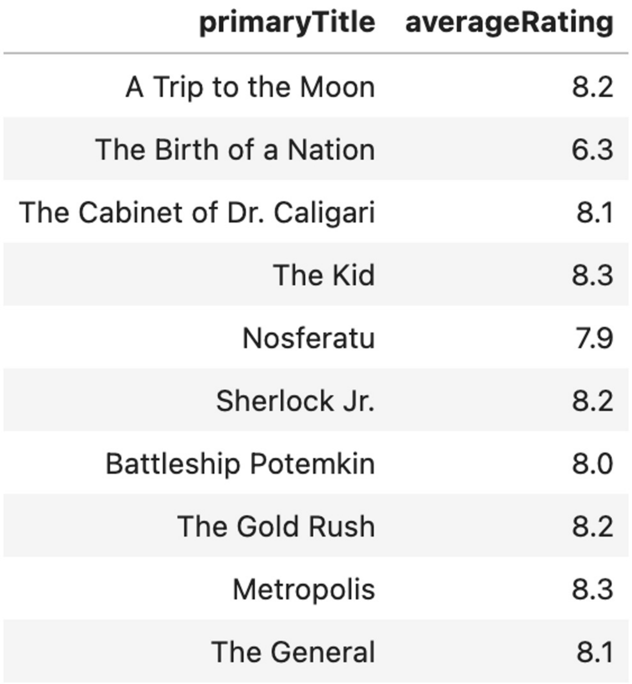
Aliasing in Joins

When working with long table names, we often create aliases that are easier to refer to (just as we did with columns yesterday).

```
SELECT primaryTitle, averageRating
FROM Title AS T INNER JOIN Rating AS R
ON T.tconst = R.tconst;
```

The AS is actually optional! We usually include it for clarity.

```
SELECT primaryTitle, averageRating
FROM Title T INNER JOIN Rating R
ON T.tconst = R.tconst;
```



primaryTitle	averageRating
A Trip to the Moon	8.2
The Birth of a Nation	6.3
The Cabinet of Dr. Caligari	8.1
The Kid	8.3
Nosferatu	7.9
Sherlock Jr.	8.2
Battleship Potemkin	8.0
The Gold Rush	8.2
Metropolis	8.3
The General	8.1

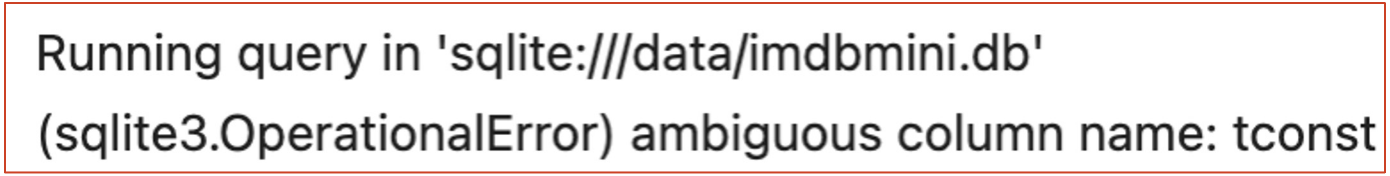
Aliasing in Joins

Why bother aliasing?

Referencing columns in the format `table_alias.column_name` avoids any ambiguity if both tables have a column with the same name.

Example: both the Title and Rating tables include a column named `tconst`

```
SELECT primaryTitle, averageRating
FROM Title INNER JOIN Rating
ON tconst = tconst;
```



```
Running query in 'sqlite:///data/imdbmini.db'
(sqlite3.OperationalError) ambiguous column name: tconst
```

Aliasing in Joins

Example: both the Title and Rating tables include a column named **`tconst`**

❌

```
SELECT primaryTitle, averageRating
FROM Title AS T INNER JOIN Rating AS R
ON tconst = tconst;
```

Should we look at the **`tconst`** column from the Title table or from the Rating table?

✅

```
SELECT primaryTitle, averageRating
FROM Title AS T INNER JOIN Rating AS R
ON T.tconst = R.tconst;
```

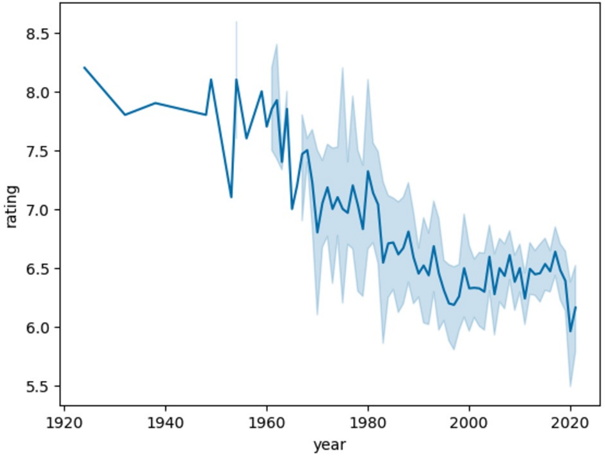
IMDB Demo

- Filtering Groups
- EDA in SQL
- Joins
- **IMDB Demo**

Demo Slides

Typical Database Workflow

- Query large amounts of data from a database using SQL. Write SQL queries to perform broad filtering and cleaning of the data
- After querying data, use **pandas** to perform more detailed analysis (visualization, modeling, etc.)


rating
year