slido

2079322

Join at slido.com
#2079322

ⓘ Click **Present with Slido** or install our Chrome extension to display joining instructions for participants while presenting.

2079322

LECTURE 21

SQL the Sequel

Expanding our SQL syntax.

Data 100/Data 200, Spring 2025 @ UC Berkeley

Narges Norouzi and Josh Grossman

Content credit: Acknowledgments

2079322

Goals for Today's Lecture

Lecture 21, Data 100 Spring 2025

Continue our tour of SQL

- Finish Basic Queries
- Grouping
- Filtering Groups
- Perform EDA in SQL
- Join Tables Together
- IMDB Demo

2079322

Finish Basic Queries

Lecture 21, Data 100 Spring 2025

- **Finish Basic Queries**
- Grouping
- Filtering Groups
- Perform EDA in SQL
- Join Tables Together
- IMDB Demo

Goal of this section

New keywords

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

2079322

By the end of this section, you will learn these new keywords!

But first, more SELECT

Recall our simplest query, which returns the full relation:

```
SELECT *
FROM Dragon;
```

table name

name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0
puff	2010	100
smaug	2011	None

`SELECT` specifies the column(s) that we wish to appear in the output. `FROM` specifies the database table from which to select data.

Every query must include a `SELECT` clause (how else would we know what to return?) and a `FROM` clause (how else would we know where to get the data?)

An asterisk (*) is shorthand for "all columns". *Let's see a bit more in our demo.*

But first, more SELECT

Recall our simplest query, which returns the full relation:

```
SELECT *
FROM Dragon;
```

table name

name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0
puff	2010	100
smaug	2011	None

We can also SELECT only a **subset of the columns**:

column expression list

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

cute	year
10	2010
-100	2011
0	2019
100	2010
None	2011

Columns selected in specified order

Aliasing with AS

2079322

To rename a `SELECT`ed column, use the `AS` keyword

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

cuteness	birth
10	2010
-100	2011
0	2019
100	2010
None	2011

An **alias** is a name given to a column or table by a programmer. Here, “cuteness” is an alias of the original “cute” column (and “birth” is an alias of “year”)

Uniqueness with `DISTINCT`

2079322

To return only unique values, combine `SELECT` with the `DISTINCT` keyword

```
SELECT DISTINCT year
FROM Dragon;
```

name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0
puff	2010	100
smaug	2011	None

year
2010
2011
2019

Notice that 2010 and 2011 only appear once each in the output.

`WHERE`: Select a rows based on conditions

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

```
SELECT name, year
FROM Dragon
WHERE cute > 0;
```

condition

name	year
hiccup	2010
puff	2010

2079322

WHERE: Select a rows based on conditions

2079322

Comparators **OR**, **AND**, and **NOT** let us form more complex conditions.

```
SELECT name, year
FROM Dragon
WHERE cute > 0 OR year > 2013;
```

condition

name	cute	year
hiccup	10	2010
puff	100	2010
dragon 2	0	2019

Check if values are contained IN a specified list

```
SELECT name, year
FROM Dragon
WHERE name IN ('hiccup', 'puff');
```

name	year
puff	2010
hiccup	2010

Strings and SQL

2079322

Strings in SQL should use **single quote:**

- **‘Hello World’** is a ***String***
- **“Hello World”** is a column name which contains a space (you can do that…)

Double quoted strings refer to columns:

- SELECT **“birth weight”** FROM patient WHERE **“first name”** = **‘Joey’**

`WHERE` with NULL Values

NULL (the SQL equivalent of NaN) is stored in a special format – we can't use the "standard" operators =, >, and <.

Instead, check if something `IS` or `IS NOT NULL`

```
SELECT name, cute
FROM Dragon
WHERE cute IS NOT NULL;
```

name	cute
hiccup	10
drogon	-100
dragon 2	0
puff	100

Always work with NULLs using the `IS` operator. NULL does not work with standard comparisons: in fact, NULL = NULL actually returns False!

ORDER BY: Sort rows

Specify which column(s) we should order the data by

```
SELECT *
FROM Dragon
ORDER BY cute DESC;
```

column

(by default, SQL orders by ascending order: **ASC**)

name	year	cute
puff	2010	100
hiccup	2010	10
dragon 2	2019	0
drogon	2011	-100
smaug	2011	None

ORDER BY: Sort rows

2079322

Specify which column(s) we should order the data by

```
SELECT *
FROM Dragon
ORDER BY year, cute DESC;
```

Can also order by multiple columns (for tiebreaks)

Sorts `year` in ascending order and `cute` in descending order. If you want `year` to be ordered in descending order as well, you need to specify `year DESC, cute DESC;`

name	year	cute
puff	2010	100
hiccup	2010	10
drogon	2011	-100
smaug	2011	None
dragon 2	2019	0

OFFSET and LIMIT?

2079322

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

Dragon

1.
```
SELECT *
FROM Dragon
LIMIT 2;
```

A.

name	year	cute
hiccup	2010	10
drogon	2011	-100

2.
```
SELECT *
FROM Dragon
LIMIT 2
OFFSET 1;
```

B.

name	year	cute
drogon	2011	-100
dragon 2	2019	0

Matching: Which query matches each relation?

What do you think the **LIMIT** and **OFFSET** keywords do?

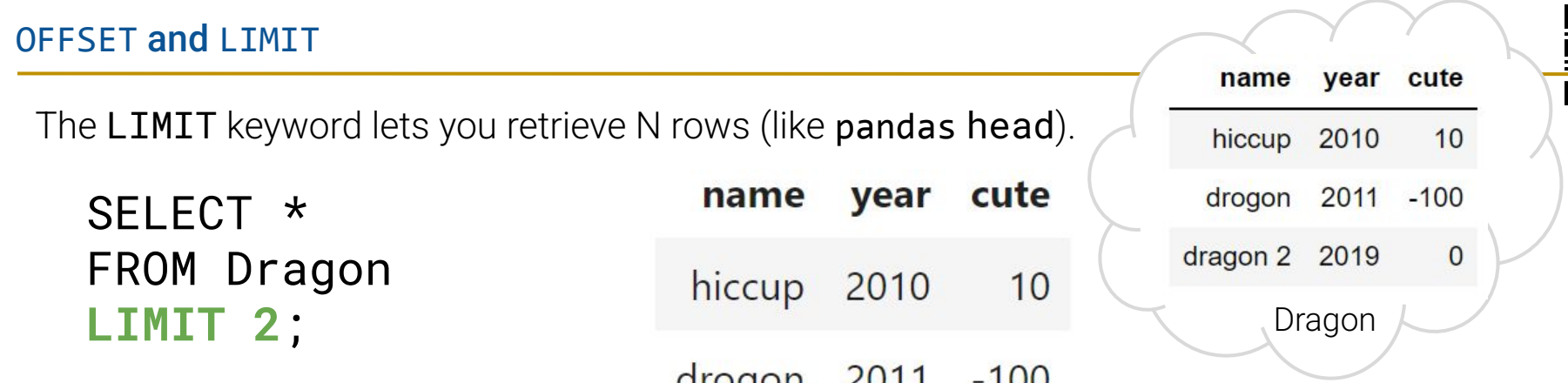
slido

2079322

Matching: Which query matches each relation?

Click **Present with Slido** or install our Chrome extension to activate this poll while presenting.

OFFSET and LIMIT



2079322

The **LIMIT** keyword lets you retrieve N rows (like **pandas head**).

```
SELECT *
FROM Dragon
LIMIT 2;
```

name	year	cute
hiccup	2010	10
drogon	2011	-100

The **OFFSET** keyword tells SQL to skip the first N rows of the output, then apply **LIMIT**.

```
SELECT *
FROM Dragon
LIMIT 2
OFFSET 1;
```

name	year	cute
drogon	2011	-100
dragon 2	2019	0

⚠️ Unless you use **ORDER BY**, there is **no guaranteed order** of rows in the relation!

New keywords

2079322

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

Summary So Far

- *All* queries must include `SELECT` and `FROM`. The remaining keywords are optional.
- By convention, use **all caps** for keywords in SQL statements.
- Use **newlines** to make code more readable.

Sampling

2079322

We can use **RANDOM** or **SAMPLE** to get a sample of the dataset.

```
%%sql
SELECT *
FROM Dragon
ORDER BY RANDOM()
LIMIT 2
```

Randomizes the entire table (reorder rows randomly) and returns two rows as requested.

```
%%sql
SELECT *
FROM Dragon USING SAMPLE reservoir(2 ROWS) REPEATABLE (100);
```

Uses a seed to randomly draw two samples from the table.
It's more efficient than ordering the entire table using **RANDOM**.

2079322

Grouping

Lecture 21, Data 100 Spring 2025

- Finish Basic Queries
- **Grouping**
- Filtering Groups
- Perform EDA in SQL
- Join Tables Together
- IMDB Demo

The Dish Table

2079322

We're ready for a more complicated table.

```
SELECT *
FROM Dish;
```

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

The Dish Table

2079322

We're ready for a more complicated table.

```
SELECT *
FROM Dish;
```

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

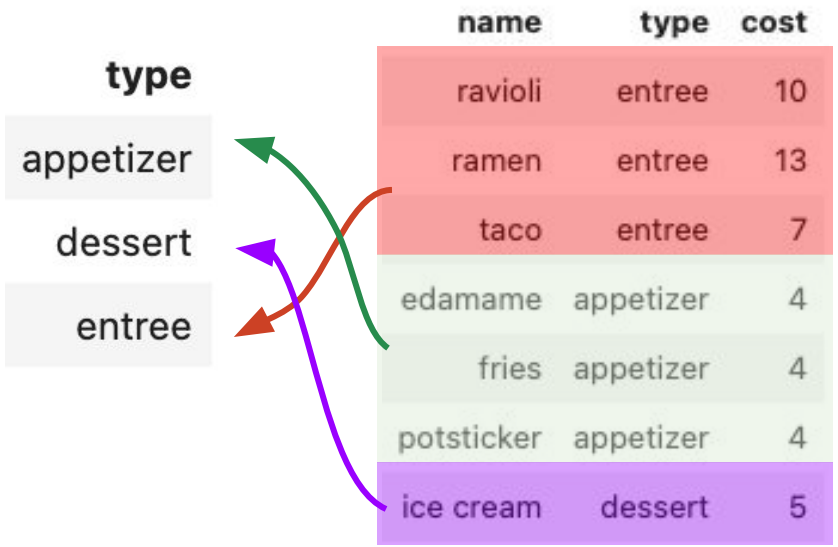
Notice the repeated dish types. What if we wanted to investigate trends across each group?

GROUP BY

2079322

`GROUP BY` is similar to pandas `groupby()`.

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



Aggregating Across Groups

2079322

Like `pandas`, SQL has **aggregate functions**: `MAX, SUM, AVG, FIRST,` etc.

For more aggregations, see: https://duckdb.org/docs/sql/aggregates.html

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

type	sum("cost")
entree	30
dessert	5
appetizer	12

Wait, something's weird…

Declarative Programming

2079322

Wait, something's weird…

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

We told SQL to SUM in our SELECT statement…

…but didn't specify the groups until GROUP BY

This is okay!

Unlike Python, SQL is a **declarative programming language**.

> *Declarative programming is a non-imperative style of programming in which programs describe their desired results without explicitly listing commands or steps that must be performed.*
>
> Wikipedia

Declarative Programming

2079322

> *Declarative programming is a non-imperative style of programming in which programs describe their desired results without explicitly listing commands or steps that must be performed.*
>
> Wikipedia

What this means to us:

- We “declare” our desired end result
- SQL handles the rest! We do *not* need to specify any logical steps for how this result should be created

We just need to follow the **SQL order of operations** with our clauses to allow SQL to parse our request. Everything else will be handled behind the scenes. (SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT) (more info)

High-level cheat sheet on order of **execution** by the SQL engine:

1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. ORDER BY

Using Multiple Aggregation Functions

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

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

2079322

Using Multiple Aggregation Functions

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

type	sum("cost")	min("cost")	max("name")
entree	30	7	taco
dessert	5	5	ice cream
appetizer	12	4	potsticker

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

2079322

The COUNT Aggregation

name	year	cute
puff	2010	100
hiccup	2010	10
drogon	2011	-100
smaug	2011	None
dragon 2	2019	0

COUNT is used to count the number of rows belonging to a group.

```
SELECT year, COUNT(cute)
FROM Dragon
GROUP BY year;
```

year	count(cute)
2010	2
2011	1
2019	1

Similar to `pandas groupby().count()`

```
SELECT year, COUNT(*)
FROM Dragon
GROUP BY year;
```

year	count_star()
2010	2
2011	2
2019	1

Similar to `pandas groupby().size()`

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

New keywords

2079322

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

Summary So Far

- 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.
- **Column Expressions may include aggregation functions (`MAX`, `MIN`, etc.)**

Filtering Groups

Lecture 21, Data 100 Spring 2025

- Finish Basic Queries
- Grouping
- **Filtering Groups**
- Perform EDA in SQL
- Join Tables Together
- IMDB Demo

Filtering Groups With `HAVING`

2079322

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;
```

Same as filter in `groupby(“type”)`**`.filter(lambda f: condition)`**

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

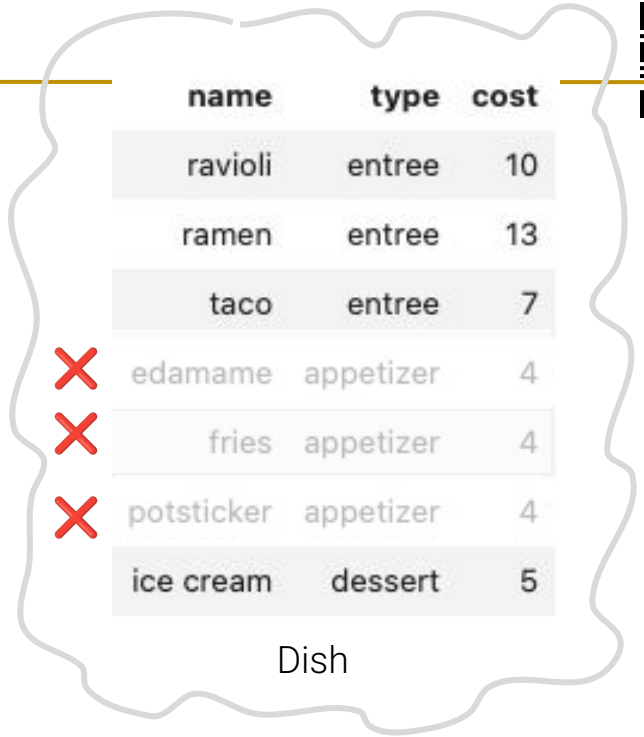
Animation: WHERE vs. HAVING

2079322

To filter:

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

WHERE precedes **HAVING**.



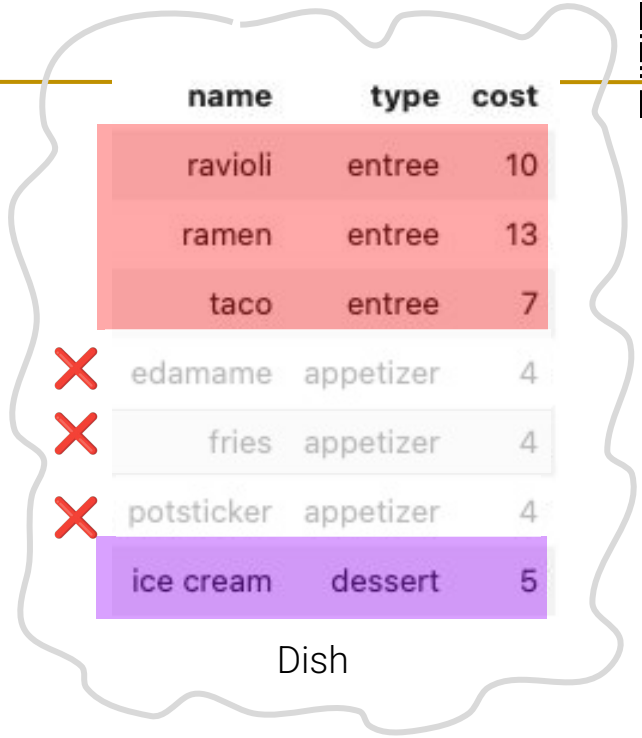

```
SELECT *
FROM Dish
WHERE cost > 4
GROUP BY type
HAVING MAX(cost) < 10;
```

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;
```

2079322

Animation: WHERE vs. HAVING

2079322

To filter:

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

WHERE precedes **HAVING**.


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

Quick Check: WHERE vs. HAVING

2079322

How many rows will be returned from the following query?

```
SELECT year, MAX(cute)
FROM Dragon
WHERE name in ('hiccup','dragon','puff')
GROUP BY year
HAVING MIN(cute) >= 0;
```

name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0
puff	2010	100
smaug	2011	None

Dragon

slido

2079322

How many rows will be returned from the following query?

Click **Present with Slido** or install our Chrome extension to activate this poll while presenting.

Quick Check: WHERE vs. HAVING

2079322

How many rows will be returned from the following query?

```
SELECT year, MAX(cute)
FROM Dragon
WHERE name in ('hiccup','dragon','puff')
GROUP BY year
HAVING MIN(cute) >= 0;
```

year	max(cute)
2010	100

Summary So Far

New keywords

2079322

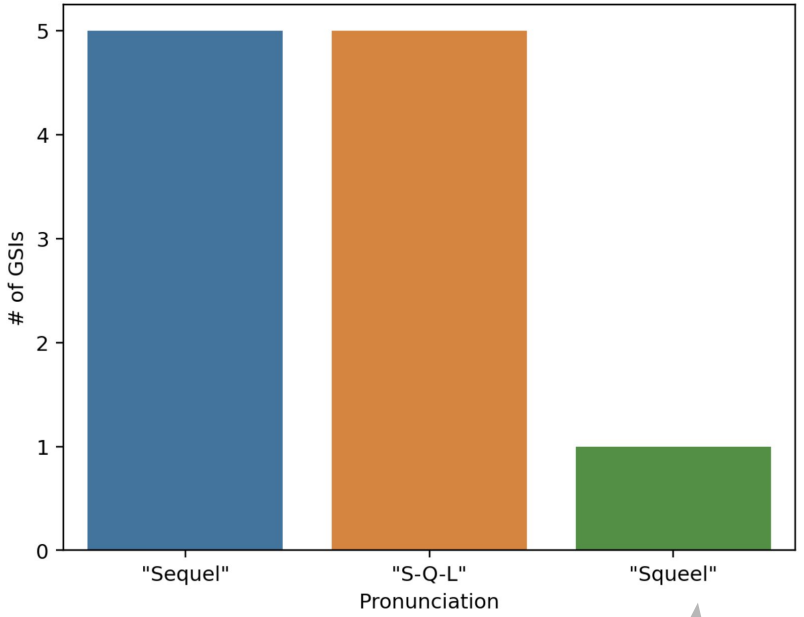
```
SELECT <column expression list>
FROM <table>
[WHERE <predicate>]
[GROUP BY <column list>]
[HAVING <predicate>]
[ORDER BY <column list>]
[LIMIT <number of rows>]
[OFFSET <number of rows>];
```

- 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.**

Interlude

“Sequel” or “S-Q-L”? Your TAs are split.

2079322



A new variant, which was a surprise to everyone involved.

2079322

Perform EDA in SQL

Lecture 21, Data 100 Spring 2025

- Finish Basic Queries
- Grouping
- Filtering Groups
- **Perform EDA in SQL**
- Join Tables Together
- IMDB Demo

The IMDB Dataset

2079322

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`

2079322

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

DuckDB and most real DBMSs also support:

SIMILAR TO '.*Star Wars.*'

(regex)

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 INTEGER)
Miss Jerry	45
The Corbett-Fitzsimmons Fight	100
Bohemios	100
The Story of the Kelly Gang	70
The Prodigal Son	90
Robbery Under Arms	None
Hamlet	None
Don Quijote	None

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

2079322

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	1894	old
movie	1897	old
movie	1905	old
movie	1906	old
movie	1907	old
movie	1907	old

2079322

Joins Tables Together

Lecture 21, Data 100 Spring 2025

- Finish Basic Queries
- Grouping
- Filtering Groups
- Perform EDA in SQL
- **Join Tables Together**
- IMDB Demo

Multidimensional Data

2079322

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

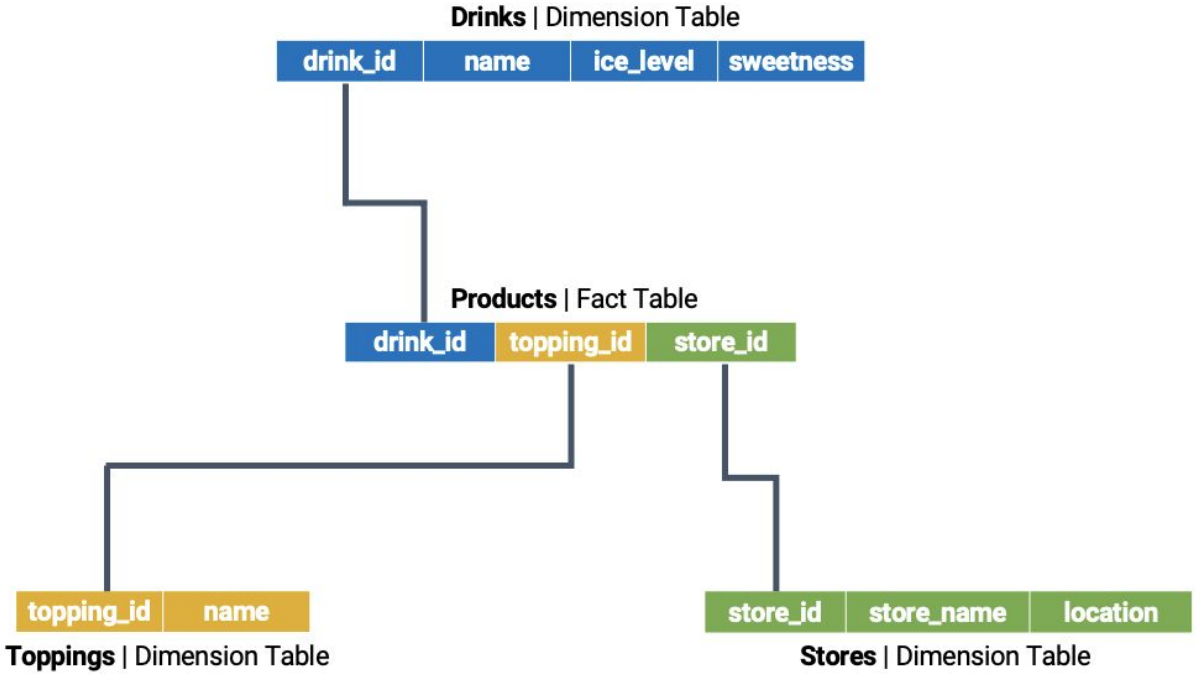
Stores | Dimension Table

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

Multidimensional Data

2079322

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



Cats

2079322

Persian

Ragdoll

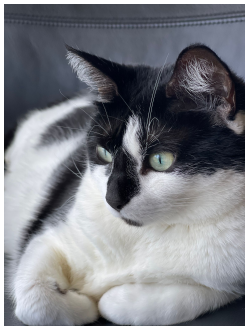
Bengal

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

2079322

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

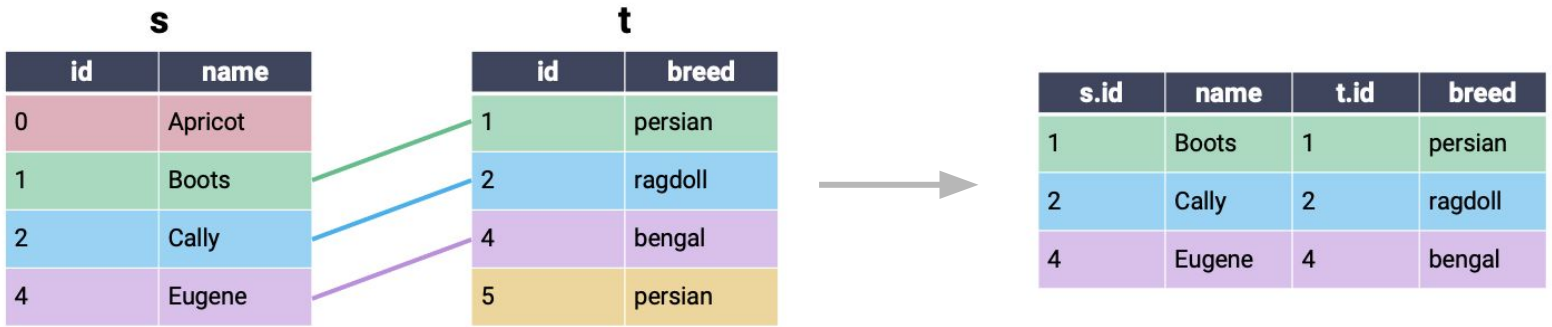
id	breed
1	persian
2	ragdoll
4	bengal
5	persian

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

Inner Join

2079322

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



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 table2
          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

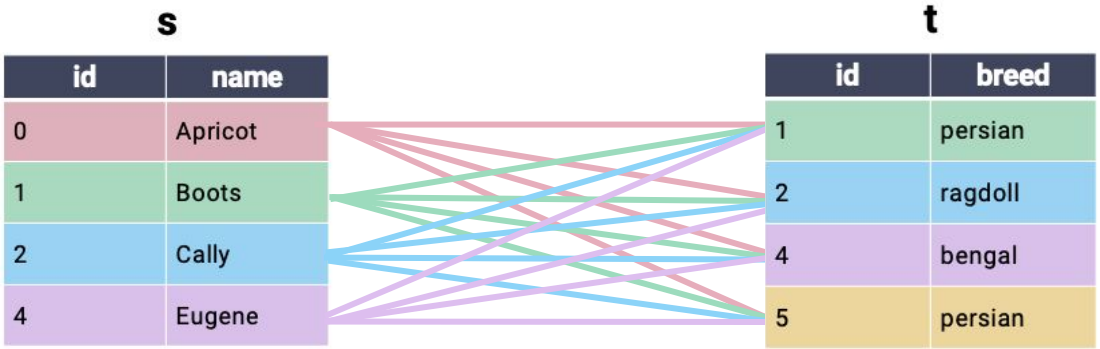
```
SELECT *
FROM s
   INNER JOIN t
   ON s.id = t.id
```

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

Cross Join

2079322

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



Cross Join

2079322

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
1	Boots	1	persian
2	Cally	2	ragdoll
4	Eugene	4	bengal

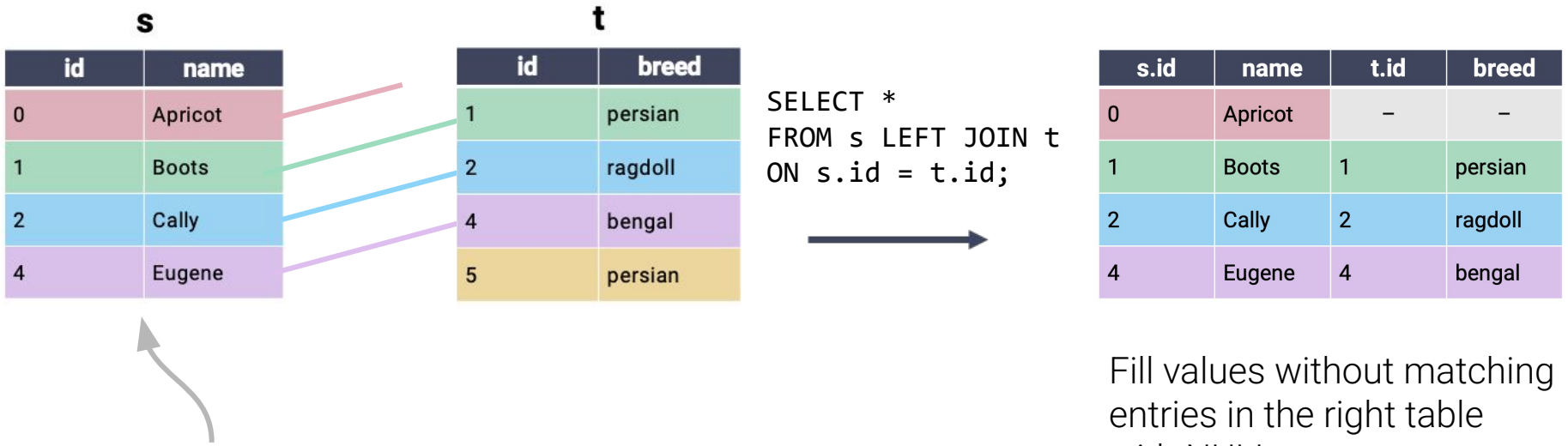
Equivalent

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.



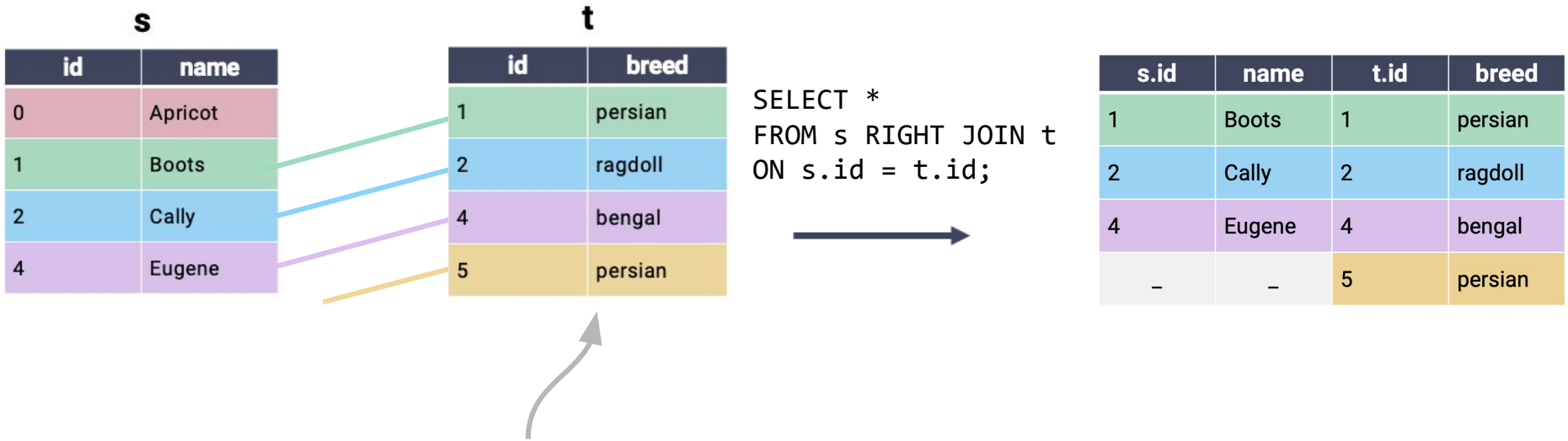
The “left table” is whichever table is referenced first in the JOIN statement.

Fill values without matching entries in the right table with NULL

Right Outer Join

2079322

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



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

Full Outer Join

2079322

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

2079322

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

```
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

2079322

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;
```

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

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;
```

Common Table Expression

2079322

Common table expression allow you to compose multiple queries.

```
WITH
table_name1 AS (
    SELECT ...
),
table_name2 AS (
    SELECT ...
)
SELECT ...
FROM
        table_name1,
        table_name2, ...
```

```
WITH
good_action_movies AS (
    SELECT *
    FROM Title T JOIN Rating R ON T.tconst = R.tconst
    WHERE genres LIKE '%Action%' AND averageRating > 7 AND numVotes > 5000
),
prolific_actors AS (
    SELECT N.nconst, primaryName, COUNT(*) as numRoles
    FROM Name N JOIN Principal P ON N.nconst = P.nconst
    WHERE category = 'actor'
    GROUP BY N.nconst, primaryName
)
SELECT primaryTitle, primaryName, numRoles, ROUND(averageRating) AS rating
FROM good_action_movies m, prolific_actors a, principal p
WHERE p.tconst = m.tconst AND p.nconst = a.nconst
ORDER BY rating DESC, numRoles DESC
LIMIT 10
```

2079322

IMDB Demo

Lecture 21, Data 100 Spring 2025

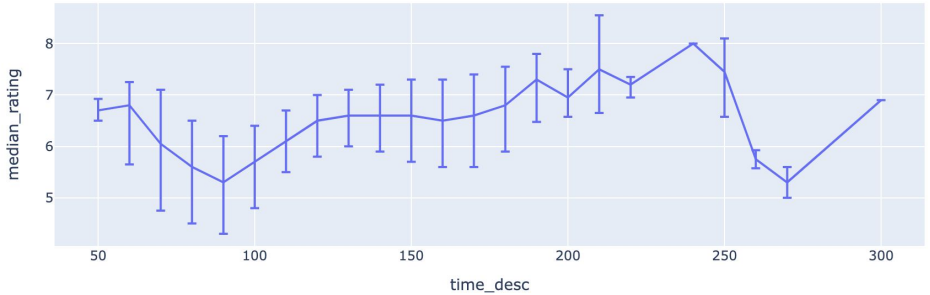
- Finish Basic Queries
- Grouping
- Filtering Groups
- Perform EDA in SQL
- Join Tables Together
- **IMDB Demo**

Demo Slides

Typical Database Workflow

2079322

- Query large amounts of data in 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.)



2079322

LECTURE 21

SQL II

Data 100/Data 200, Spring 2025 @ UC Berkeley

Narges Norouzi and Josh Grossman

Content credit: Acknowledgments