</talentlabs>

CHAPTER 1

Intro to Relational Databases

</talentlabs>

AGENDA

- Course Introduction
- Intro to Databases
- Building First Database with Excel
- The Need for an Professional Database
- Different Types of Databases
- Interacting with Databases
- Database for the Course

Course Introduction

</talentlabs>

Course Objectives

- ✅ Get to operate a wide range of SQL database, in particular SQLite
- ✅ Basic proficiencies in SQL query language
- ✅ SQL aggregation using GROUP BY statement
- ✅ Advanced queries with logics and various functions
- ✅ Identify primary keys and foriegn keys
- ✅ SQL JOINs to combine tables with different dimensions
- ✅ SQL tables management using SQL statements
- ✅ ERD diagram to visualize SQL table linkages

</talentlabs>

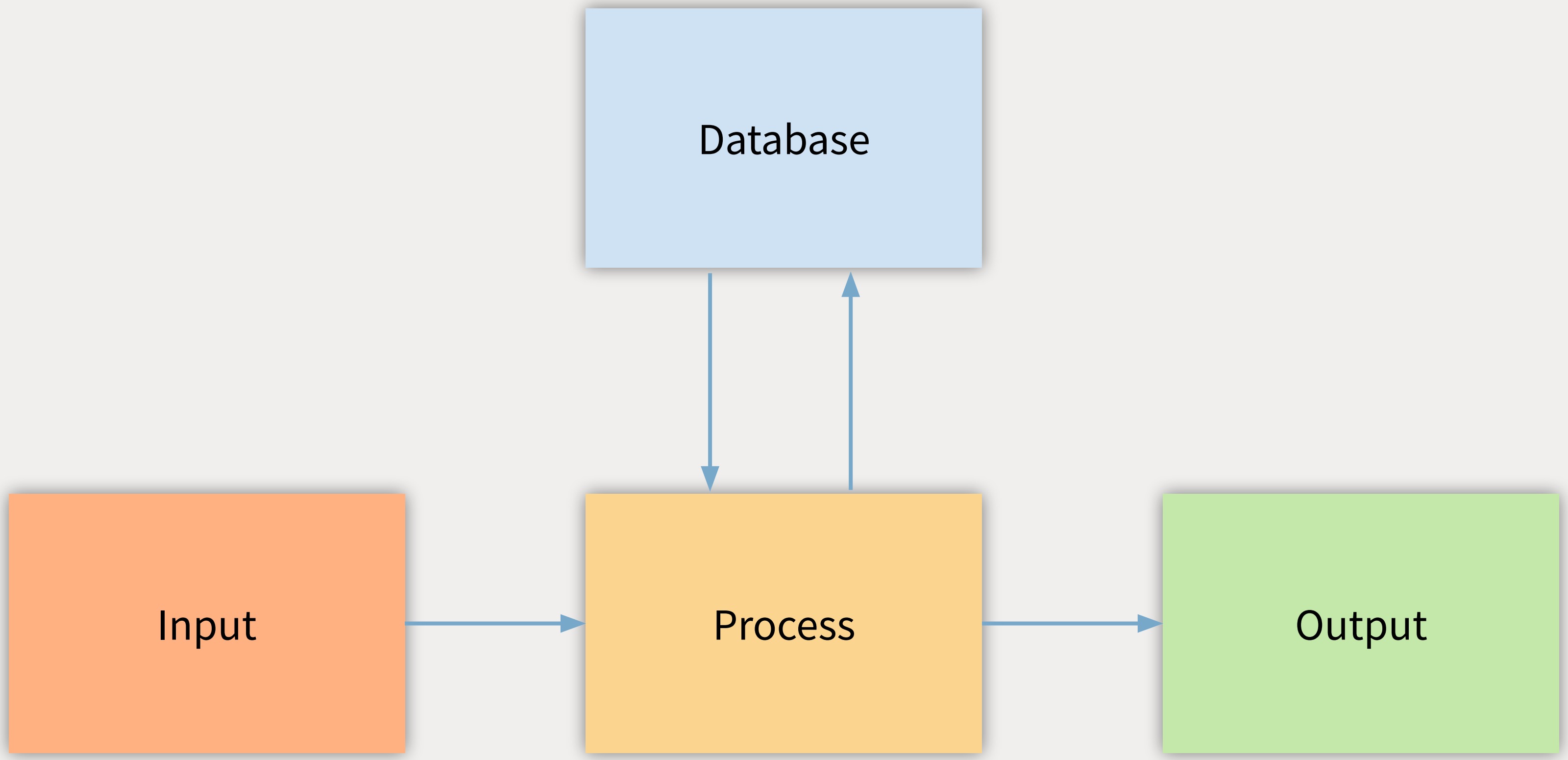
Database 101

What is Database?

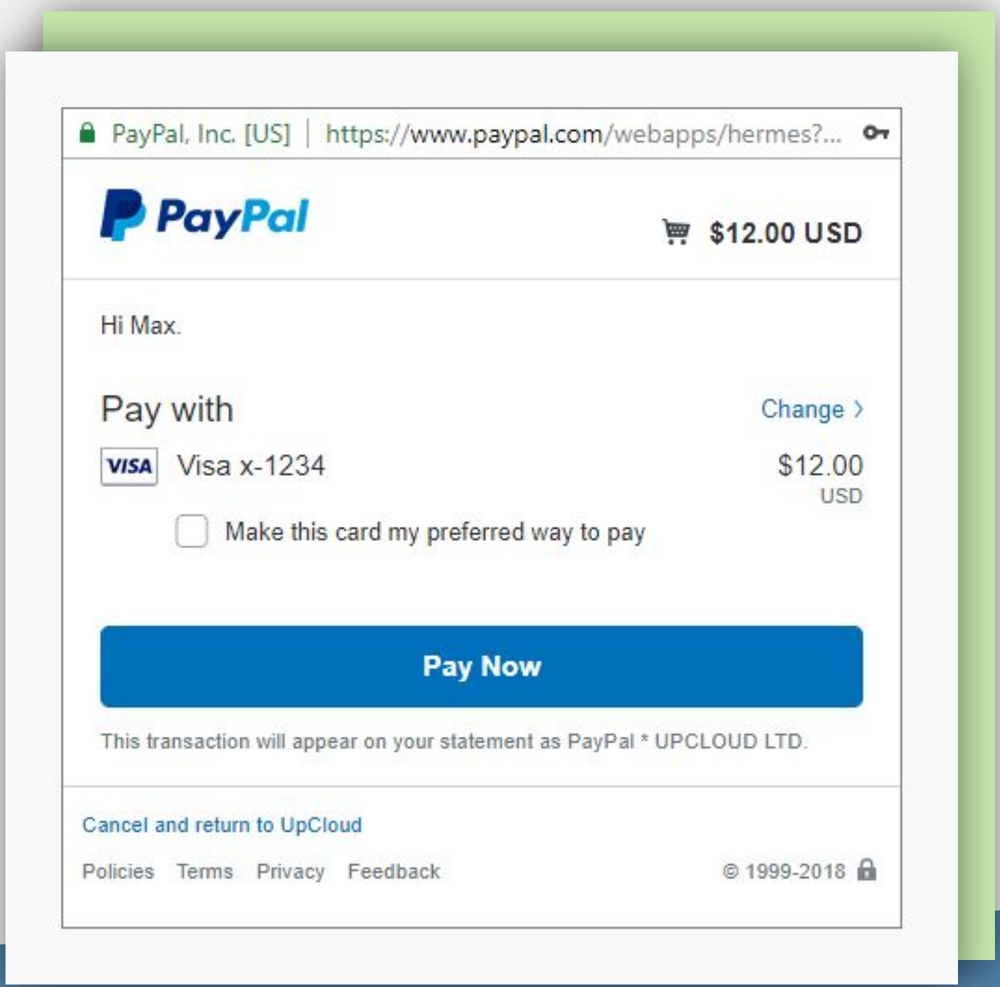
A database is a **collection of information** that is **organized** so that it can be **easily accessed, managed and updated**. Computer databases typically contain aggregations of data records or files, containing information about sales transactions or interactions with specific customers.

-- Whatis.com

Elements of a SYSTEM



Which Applications use Database?



PayPal, Inc. [US] | https://www.paypal.com/webapps/hermes?...

PayPal $12.00 USD

Hi Max.

Pay with Change ›

VISA Visa x-1234 $12.00 USD

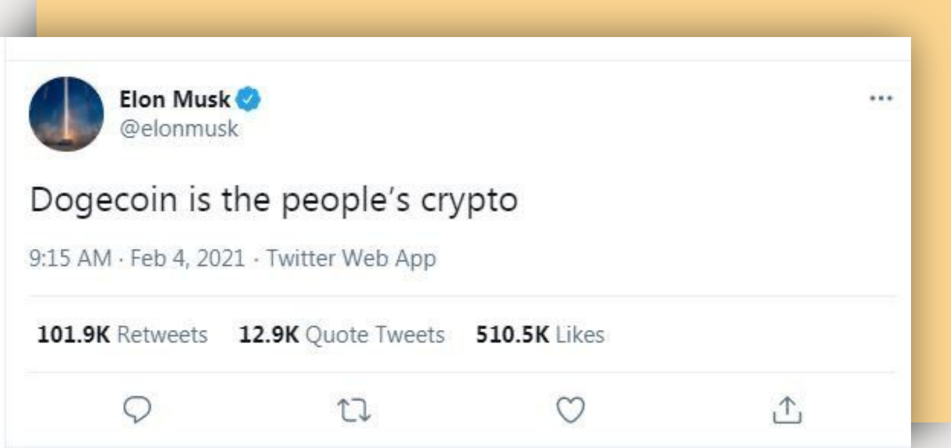
□ Make this card my preferred way to pay

Pay Now

This transaction will appear on your statement as PayPal * UPCLOUD LTD.

Cancel and return to UpCloud

Policies Terms Privacy Feedback © 1999-2018



Elon Musk
@elonmusk

Dogecoin is the people's crypto

9:15 AM · Feb 4, 2021 · Twitter Web App

101.9K Retweets **12.9K** Quote Tweets **510.5K** Likes

</talentlabs>

Why Do We Need Databases?

User
Status

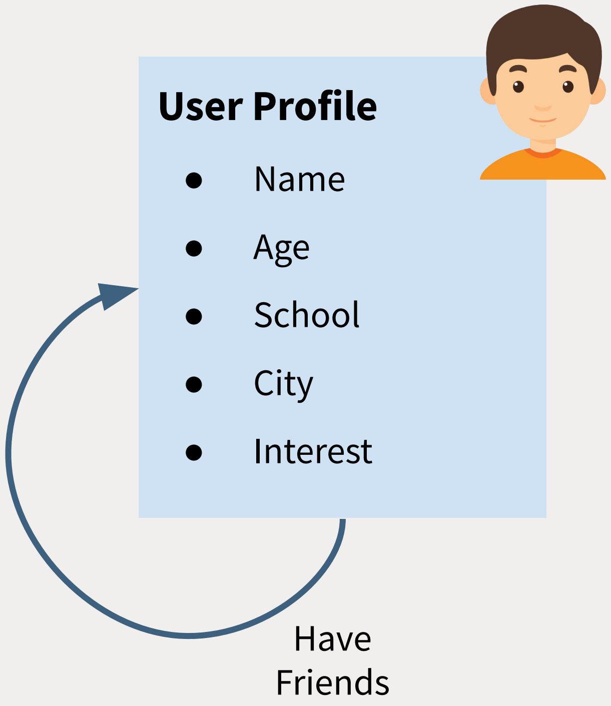
User
History

User
Data

Website
Content

Building our First Database with Excel

Imagine we are Facebook...

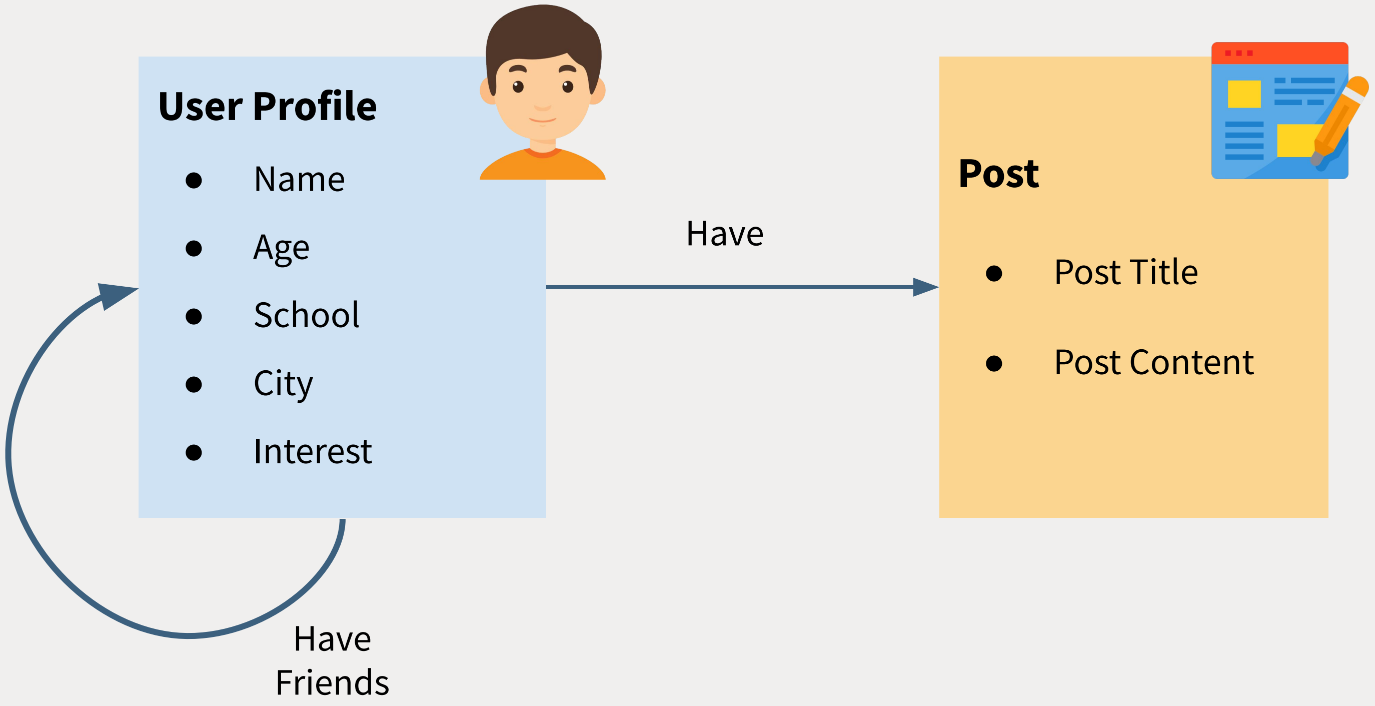


User Profile

- Name
- Age
- School
- City
- Interest

Have Friends

Imagine we are Facebook...



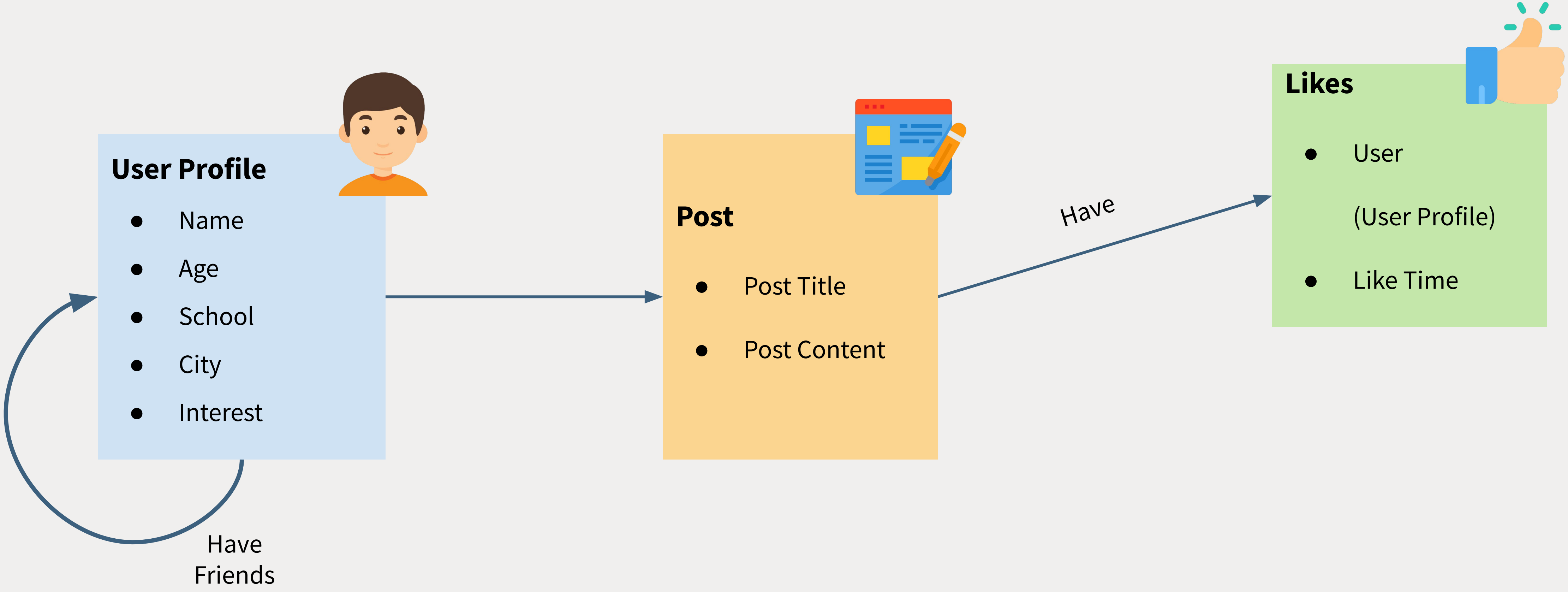
User Profile

- Name
- Age
- School
- City
- Interest

Post

- Post Title
- Post Content

Imagine we are Facebook...



User Profile

- Name
- Age
- School
- City
- Interest

Have Friends

Post

- Post Title
- Post Content

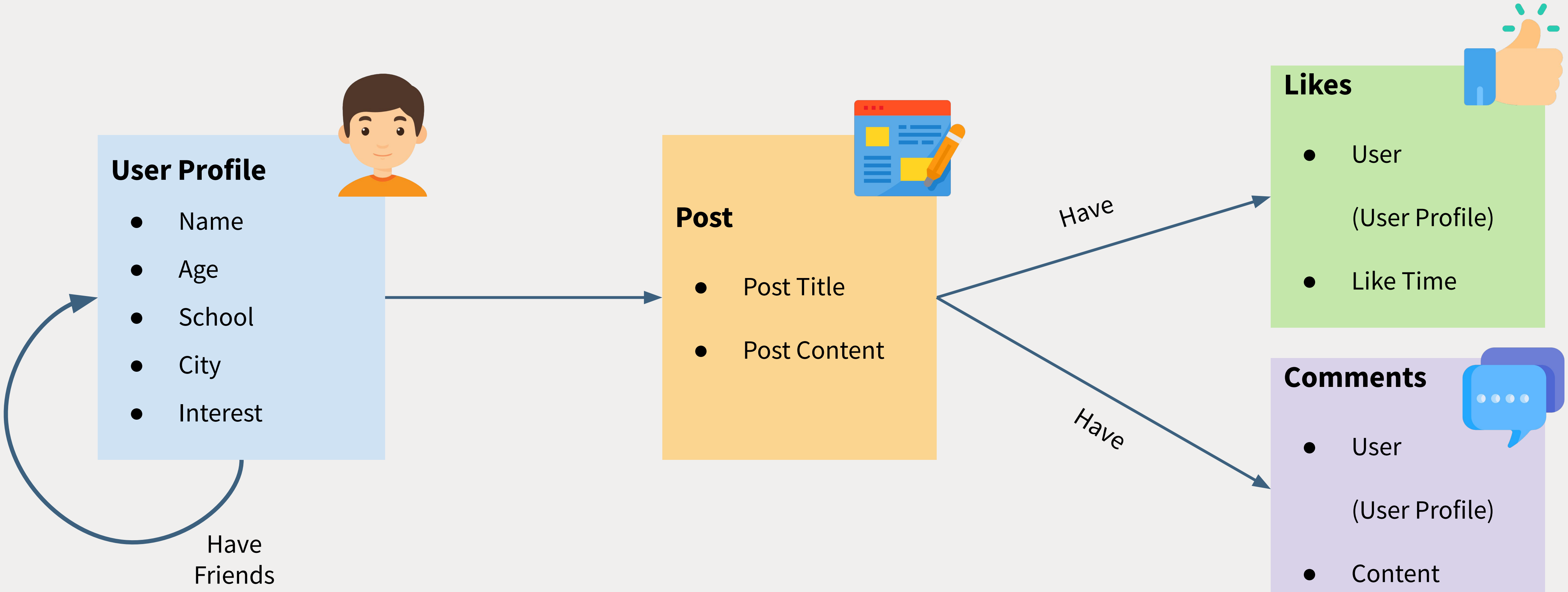
Have

Likes

- User

  (User Profile)
- Like Time

Imagine we are Facebook...



User Profile

- Name
- Age
- School
- City
- Interest

Have Friends

Post

- Post Title
- Post Content

Have

Have

Likes

- User (User Profile)
- Like Time

Comments

- User (User Profile)
- Content

Let’s get STARTED with Excel!

- Building a Facebook Database with Excel!
- Three important concepts:
  - **Table** - Represent a physical concept
  - **Column/Field** - Represent a property
  - **Row** - Represent a record

</talentlabs>

Imagine we are Facebook...

🤔 The data is too complex to just write into Excel

🏢 We have to store the data somewhere - every user’s hard drive?

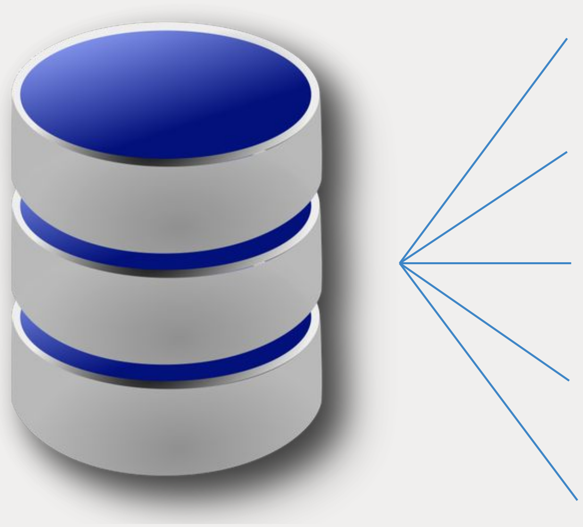
⚙️ There must be a system to manage the data

The Need for a Professional Database

Excel as Database? Great START, but...

- ❌ Too easy to make changes
- ❌ Hard to keep track of changes
- ❌ Cannot be accessed by multiple people at the same time
- ❌ Limited row count - 1,048,576 rows (for xlsx file)
- ❌ Cannot locate a single record easily
- ❌ Cannot create summary to large amount of data easily

Additional Database Features



- Backup Mechanisms
- Simultaneous Connections
- Data Integrity Features
- Scalability
- Speed

Different Types of Databases

Types of Databases

- For professional databases, there are two main types:

SQL

(Relational Database)

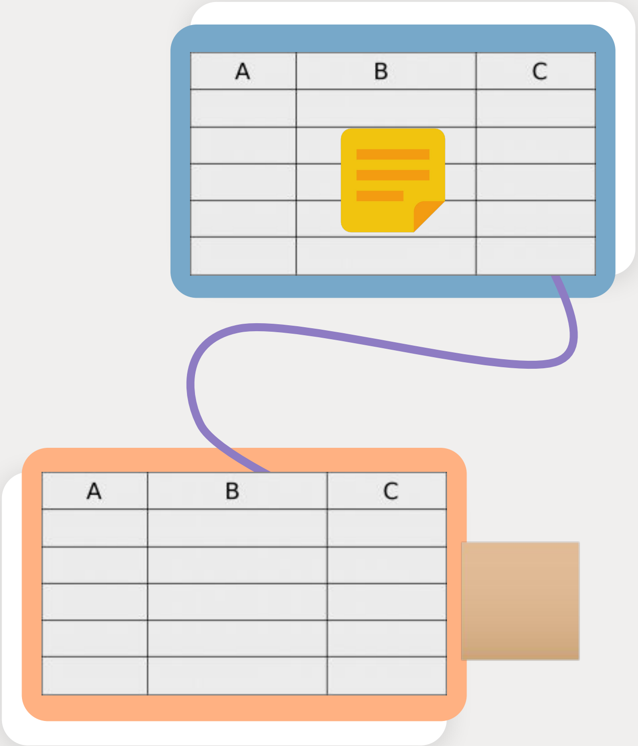
NoSQL

(Document Store)

- Focus of this course: **Relational Database/SQL**

Relational Databases/SQL

- A relational database has a clear data structure (in tables, rows, and columns)
- Each table on the database **can be linked** to each other
    - **Example**: On Facebook, a “Post” table can be linked to a “Like” table as “posts can have likes”
- SQL queries can retrieve or summarize data from tables



Examples of Relational Databases

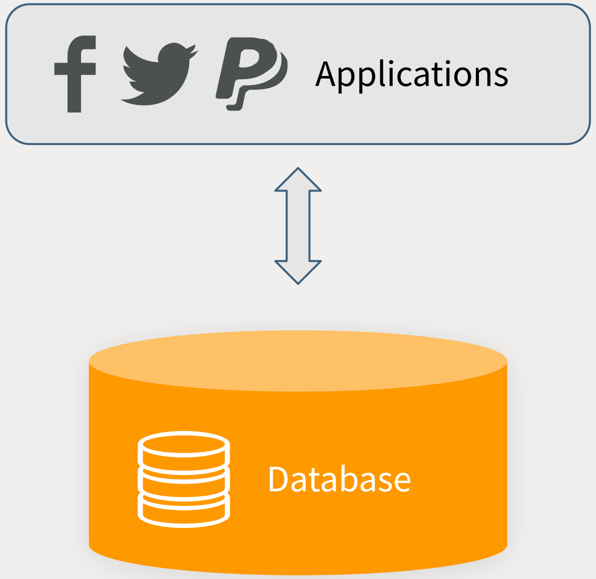
SQL DB (Relational DB)

MySQL
Microsoft SQL Server
SQLite
PostgreSQL

Interacting with Databases

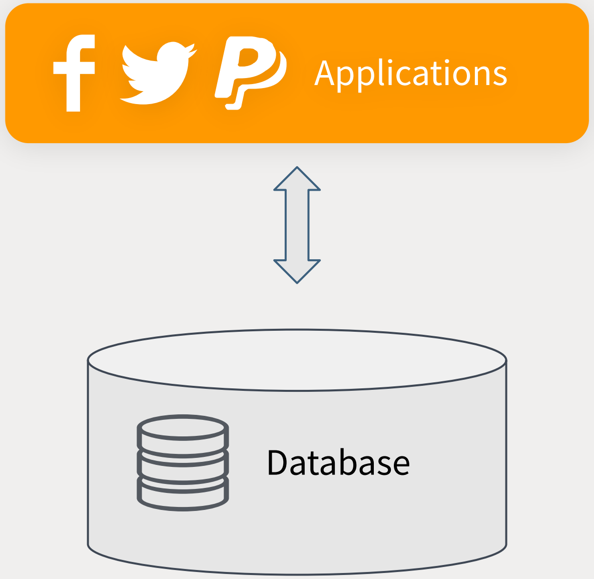
</talentlabs>

Role of a Database



- A database is both
    - data storage space, and
    - interface to pull/query the data
- Applications (e.g. apps, web pages) talk to the database to pull data
- Once the data is retrieved, the data is then displayed in the user interfaces (e.g. apps, web pages)

Role of an Application



- Apart from retrieving data, an application can also creates/updates user records according to some business logic
- Application will send instructions to the database and database will store the data accordingly

Accessing a Database

Using Database clients - web interface or application-based **(a software)**

Using programming interfaces **(a program)**

Using command line **(seldomly used)**

Database for the Course

We’ll be using SQLite!

Why SQLite?

- Run locally as a embedded database
  - **No installation** is needed
  - Other SQL databases require a server to run on (Complicated setup)
- **Can be embedded** into a program
  - WhatsApp chat history is maintained on a SQLite database on your mobile phone!
- We will walk through the database setup in the next chapter

</talentlabs>

CHAPTER 2

Environment Setup

</talentlabs>

AGENDA

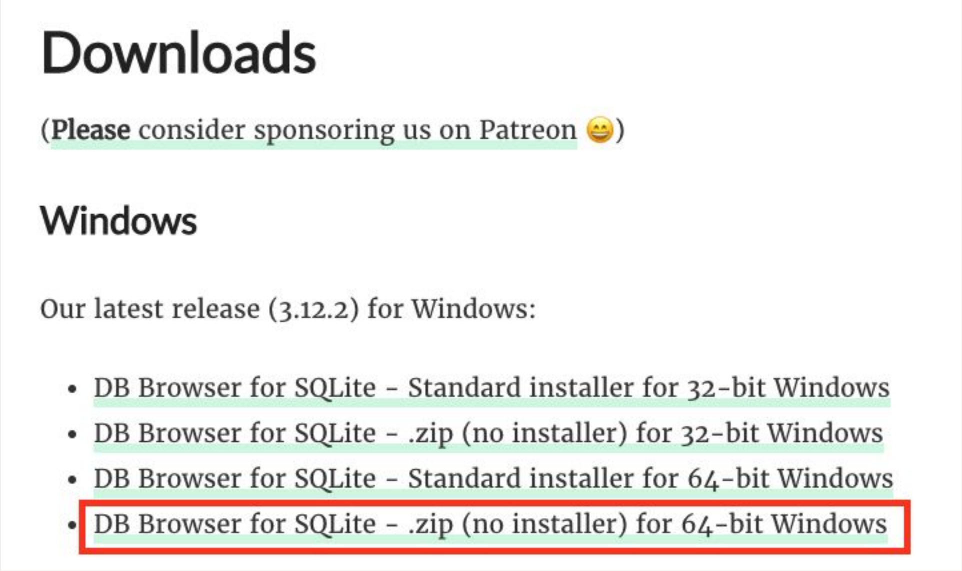
- Install DB Browser for SQLite
- Load our first database locally
- Execute our first SQL query

Install DB Browser for SQLite

</talentlabs>

DB Browser for SQLite

- A software to create, modify, search, and query on a SQLite database
- Download link - https://sqlitebrowser.org/dl/
- Use 64 bit and "no installer" version for Windows
- We'll mainly use the software for opening a database and making queries



Downloads

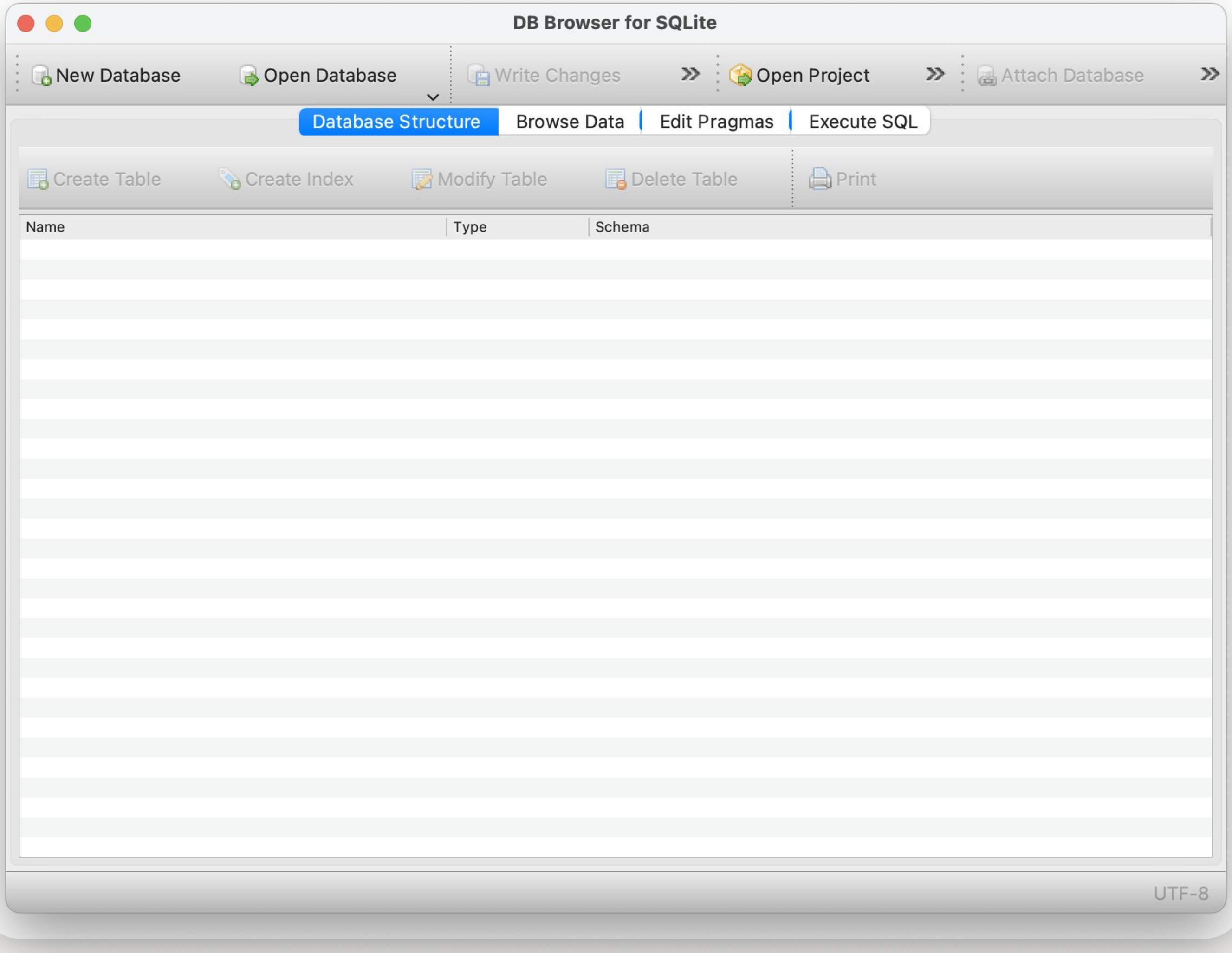
(**Please** consider sponsoring us on Patreon 😄)

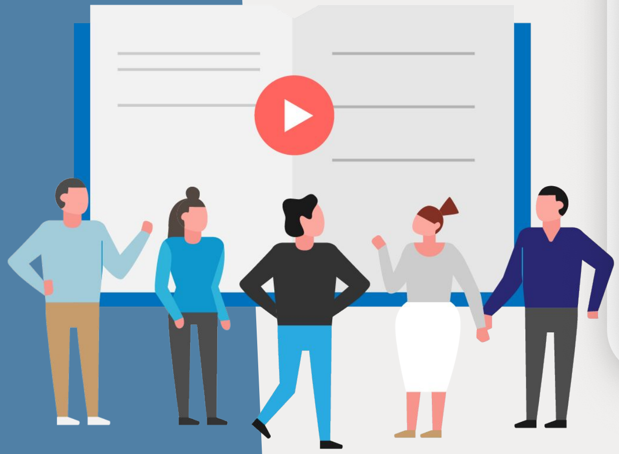
Windows

Our latest release (3.12.2) for Windows:

- DB Browser for SQLite – Standard installer for 32-bit Windows
- DB Browser for SQLite – .zip (no installer) for 32-bit Windows
- DB Browser for SQLite – Standard installer for 64-bit Windows
- DB Browser for SQLite – .zip (no installer) for 64-bit Windows

DB Browser for SQLite





Load our First Database

</talentlabs>

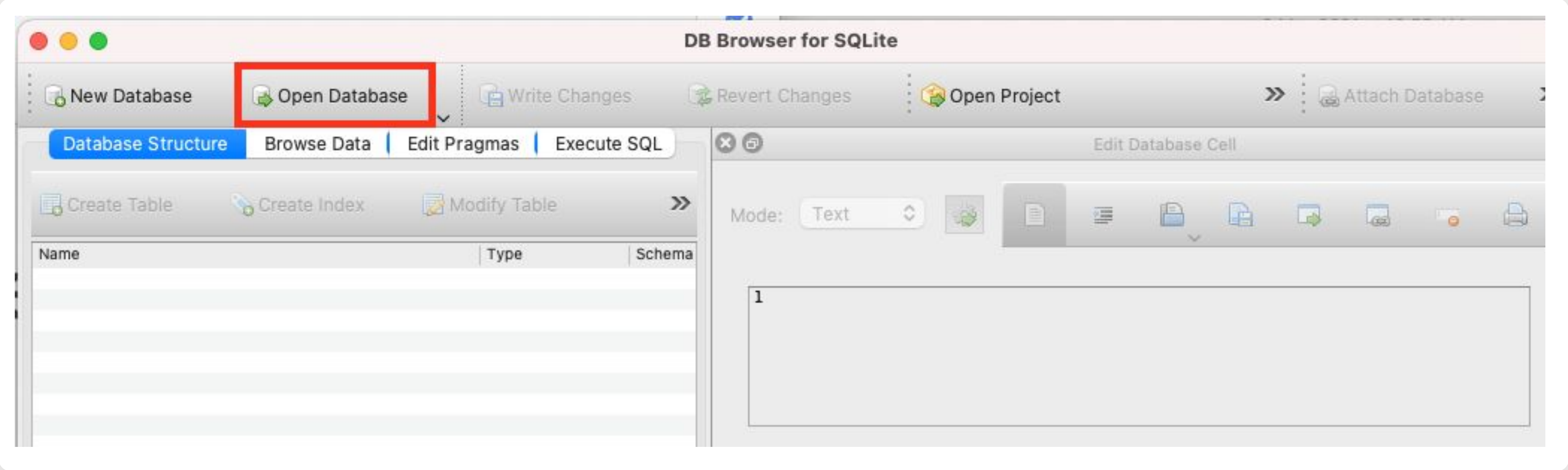
Movies Database

- From your Assignment page, please download and unzip the file - movies.zip
- You should have a folder called “movies”
- Inside the folder there should have a file called “movies.db”
- This is the file we will mainly work on throughout this course

</talentlabs>

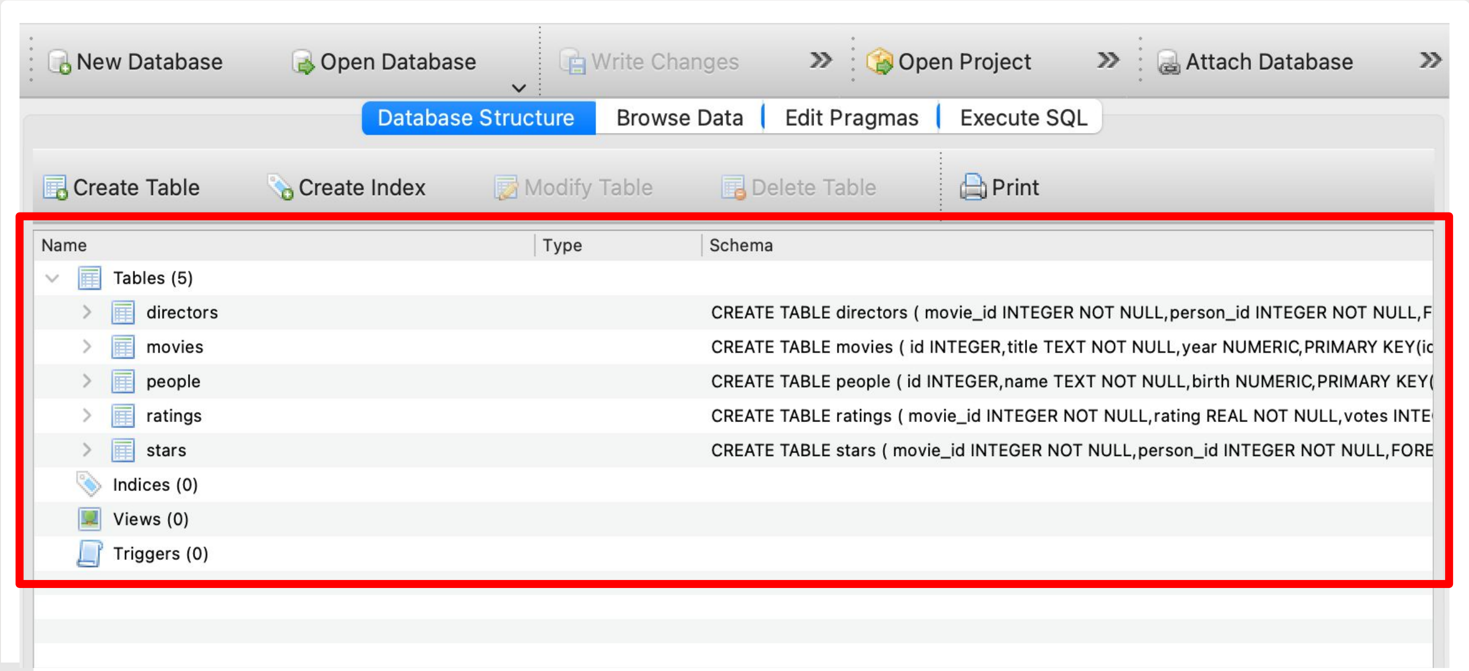
Movies Database

- Click the button “Open Database”
- Select and open the file “movies.db” which you’ve just unzipped
- You should see there are 5 tables inside this movies database



</talentlabs>

Movies Database

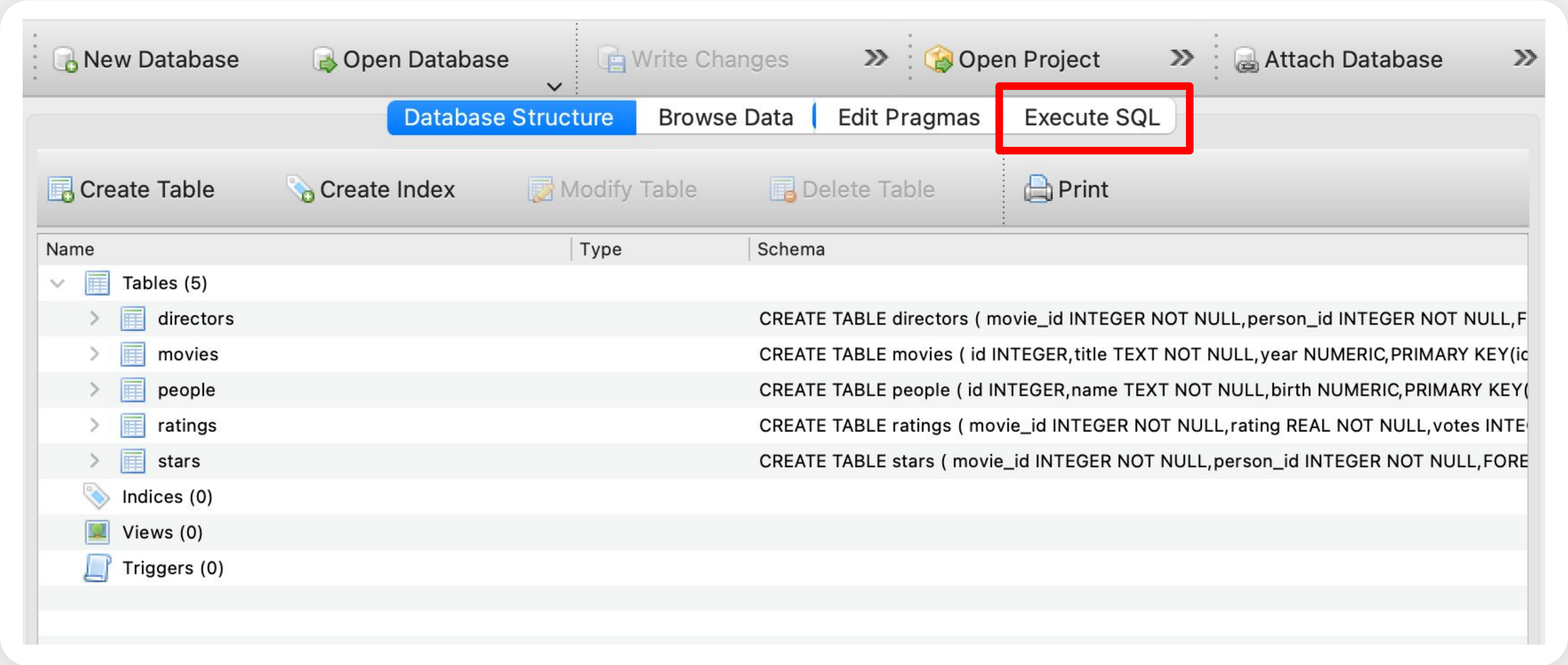


Name	Type	Schema
Tables (5)		
directors		CREATE TABLE directors (movie_id INTEGER NOT NULL,person_id INTEGER NOT NULL,F
movies		CREATE TABLE movies (id INTEGER,title TEXT NOT NULL,year NUMERIC,PRIMARY KEY(id
people		CREATE TABLE people (id INTEGER,name TEXT NOT NULL,birth NUMERIC,PRIMARY KEY(
ratings		CREATE TABLE ratings (movie_id INTEGER NOT NULL,rating REAL NOT NULL,votes INTE
stars		CREATE TABLE stars (movie_id INTEGER NOT NULL,person_id INTEGER NOT NULL,FORE
Indices (0)		
Views (0)		
Triggers (0)		

Execute Our First SQL Query

Step 1: “Execute SQL” Tab

Click the tab “Execute SQL”



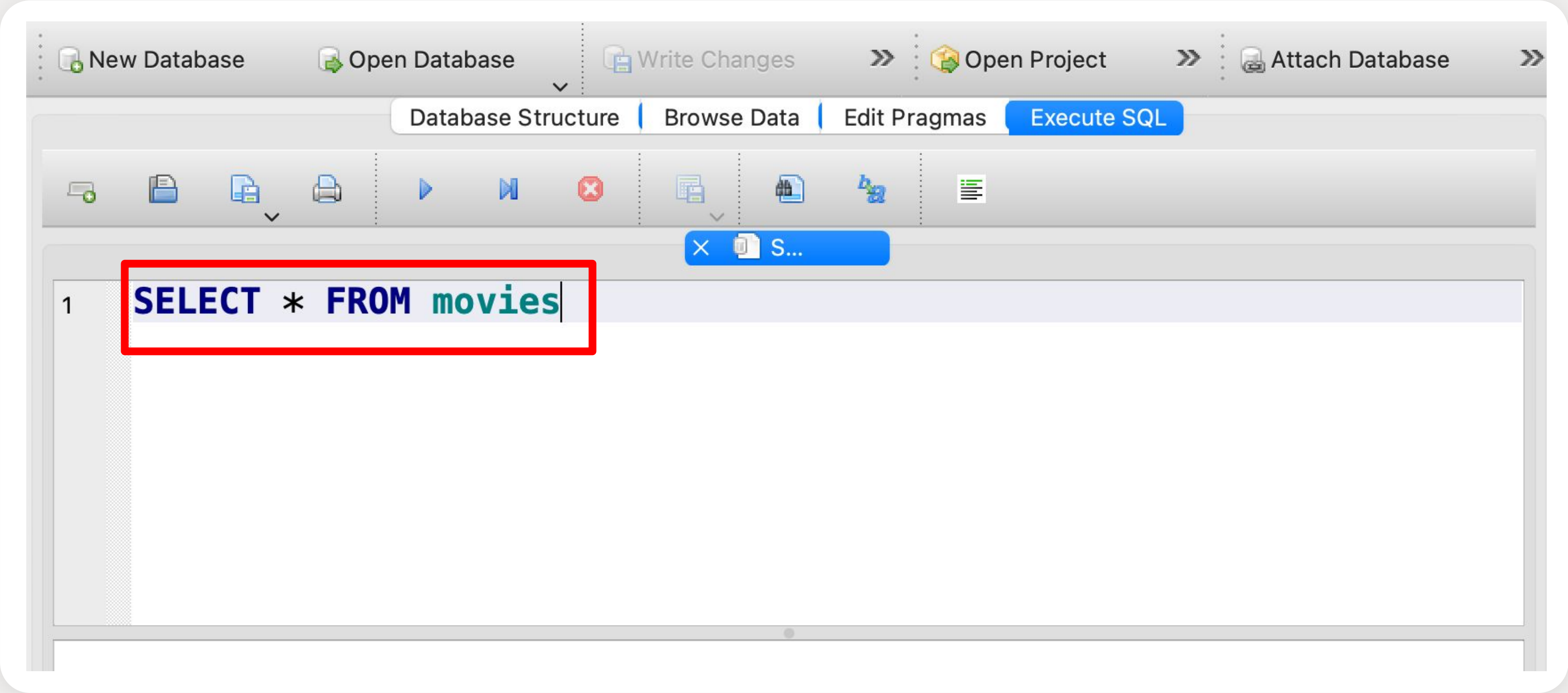
Name	Type	Schema
Tables (5)		
directors		CREATE TABLE directors (movie_id INTEGER NOT NULL,person_id INTEGER NOT NULL,F
movies		CREATE TABLE movies (id INTEGER,title TEXT NOT NULL,year NUMERIC,PRIMARY KEY(ic
people		CREATE TABLE people (id INTEGER,name TEXT NOT NULL,birth NUMERIC,PRIMARY KEY(
ratings		CREATE TABLE ratings (movie_id INTEGER NOT NULL,rating REAL NOT NULL,votes INTE
stars		CREATE TABLE stars (movie_id INTEGER NOT NULL,person_id INTEGER NOT NULL,FORE
Indices (0)		
Views (0)		
Triggers (0)		

Step 2: Type in the SQL Query

Input the SQL query in the box.

You may try the below query:

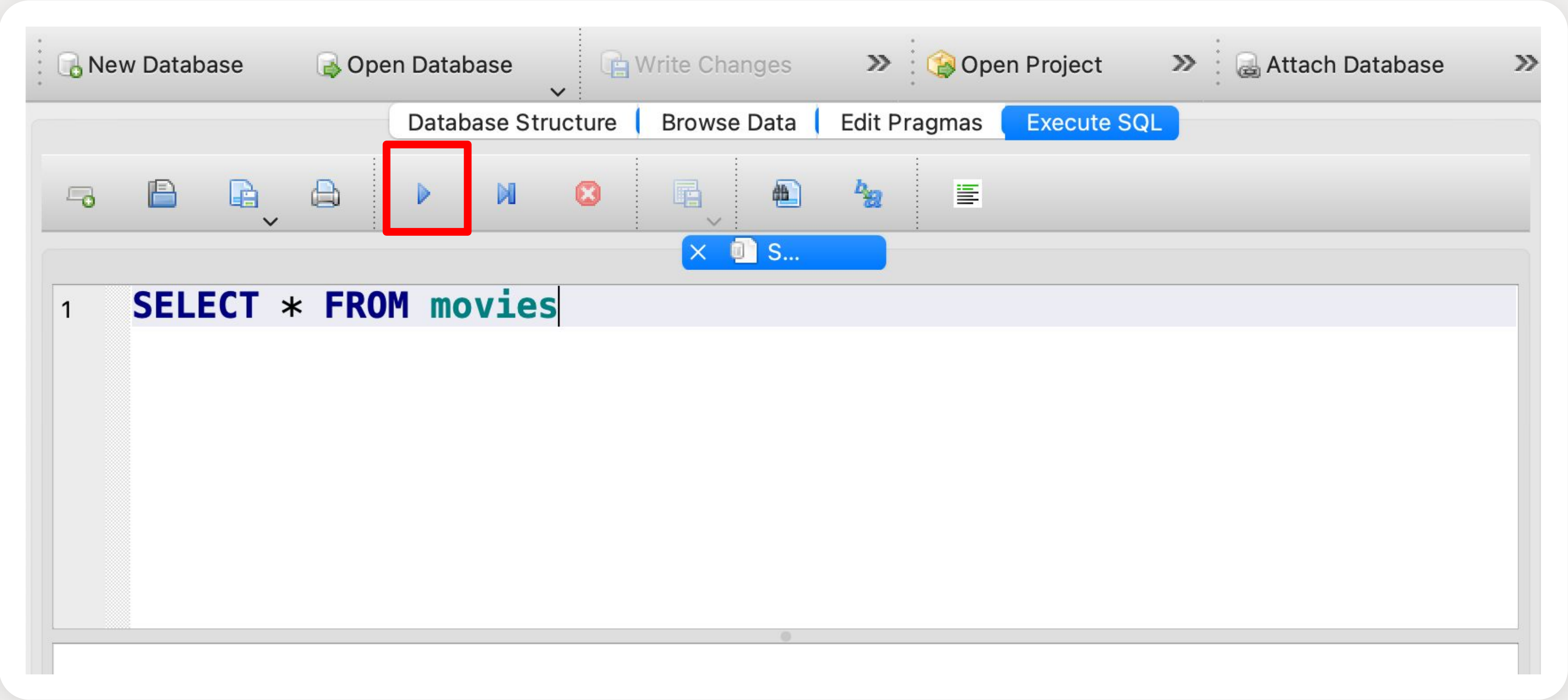
```
SELECT * FROM movies
```



```
SELECT * FROM movies
```

Step 3: Click the Run Button

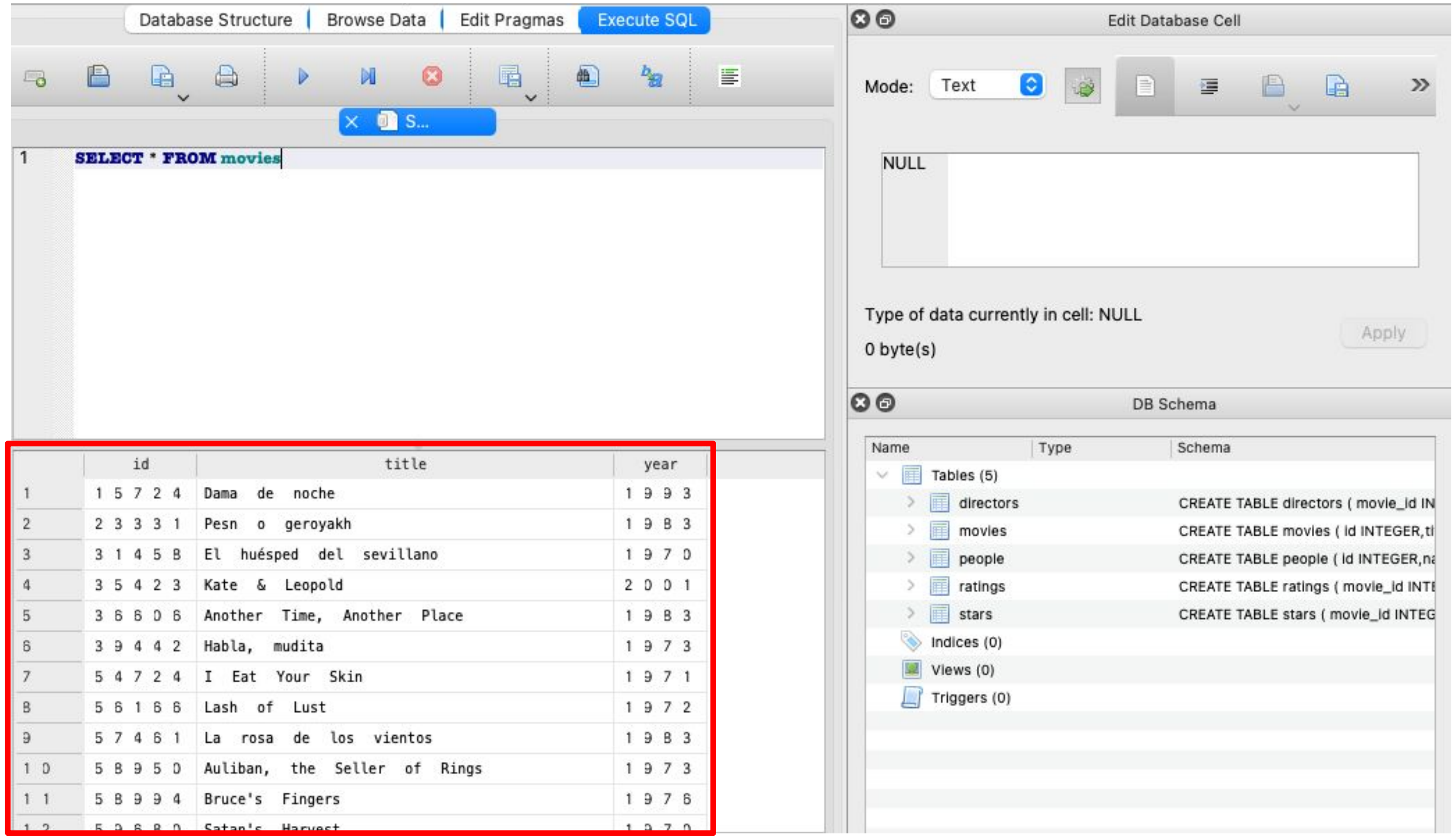
Click the Run button



```
SELECT * FROM movies
```

</talentlabs>

Step 4: Voila! Check the Results



```
SELECT * FROM movies
```

	id	title	year
1	15724	Dama de noche	1993
2	23331	Pesn o geroyakh	1983
3	31458	El huésped del sevillano	1970
4	35423	Kate & Leopold	2001
5	36606	Another Time, Another Place	1983
6	39442	Habla, mudita	1973
7	54724	I Eat Your Skin	1971
8	56166	Lash of Lust	1972
9	57461	La rosa de los vientos	1983
10	58950	Auliban, the Seller of Rings	1973
11	58994	Bruce's Fingers	1976
12	59680	Satan's Harvest	1970

Mode: Text

NULL

Type of data currently in cell: NULL

0 byte(s)

Name	Type	Schema
Tables (5)		
directors		CREATE TABLE directors (movie_id IN
movies		CREATE TABLE movies (id INTEGER,ti
people		CREATE TABLE people (id INTEGER,na
ratings		CREATE TABLE ratings (movie_id INTE
stars		CREATE TABLE stars (movie_id INTEG
Indices (0)		
Views (0)		
Triggers (0)		

But what we just did?

- We just listed out all the movies from the movie database
- We'll go through the syntax (or meaning) of writing SQL queries in coming lectures.

</talentlabs>

CHAPTER 3

SQL Foundations

</talentlabs>

AGENDA

- CRUD Concepts
- Data Type

CRUD

What is CRUD?

C

Create

Create a new record and save in the database. Or it can refers to creating a new table

R

Read

Get some records from the database.

There are no changes to the data in this operation

U

Update

Updating one or more existing records in the database.

D

Delete

Removing some data from the database table

</talentlabs>

SQL Keywords

- Are reserved words of SQL queries that serve special functions
- Some most frequently used common keywords are:

SELECT **CREATE** **DELETE**

UPDATE **INSERT INTO** **ORDER BY**

DESC **ASC** **LIMIT**

- Can you guess the functions of these keywords?

Action Keywords

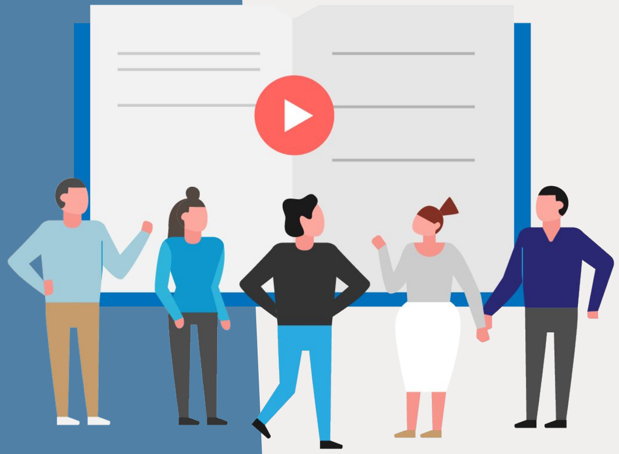
SQL Keyword	Function	Related CRUD operation
CREATE	Create a new data table	CREATE
INSERT INTO	Insert new records in an existing data table	CREATE
SELECT	Obtain data from a table(s)	READ
UPDATE	Update record(s) in a table	UPDATE
DELETE	Delete records(s) from a table	DELETE

</talentlabs>

Do you remember our first query?

```
SELECT * FROM movies
```

- It is a Read operation in terms of CRUD!
- Apart from reading the data of the whole table, we can select specific data with condition or even summarize the data by aggregation!
- We will cover these operations in coming chapters



SQL Data Types

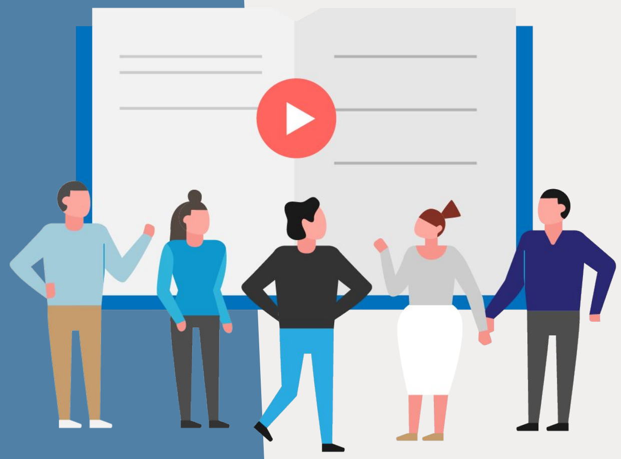
Data Types

- Data types defined the type of data to be stored in a column.
- SQL databases generally offer the below categories of data type

TEXT **INTEGER** **DECIMAL**

BOOLEAN **DATE & TIME**

- Actual data types and naming vary in different databases



Examples of different data types

Type	Example Data Types
INTEGER	1, 2, 3, 4, 5
DECIMAL	1.38, 3.49, 999.211
TEXT	“Harry Potter”, “Fast and Furious”
DATETIME	2016-08-30 18:47:56.235
BOOLEAN **(There are no boolean type in SQLite)**	0 or 1 (SQLite use Integer) True or False (Other Databases)

</talentlabs>

CHAPTER 4

Basic SELECT Queries

</talentlabs>

AGENDA

- SELECT Basic Structure
- SELECT DISTINCT
- LIMIT and ORDER BY

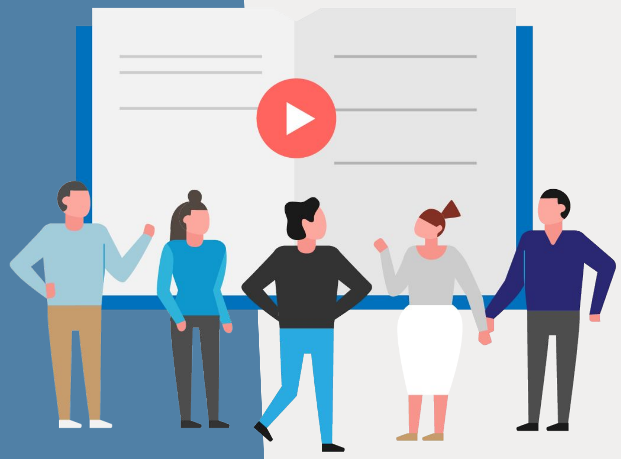
SELECT Basic Structure

SQL Syntax

- SQL query can be easily understood as the query syntax reads fairly like human language
- Here is the simplified structure of a SQL query

```
SELECT columns    FROM table    WHERE condition(s)
GROUP BY column(s)    ORDER BY column(s)    LIMIT n
```

- SQL keywords are typed in capital letters for better readability. However in most cases SQL queries are case insensitive.



SQL SELECT FROM

- Select some columns from a table:

```
SELECT col_name1, col_name2 FROM a_table
```

- In case of getting all the column of a table, use * to represent the columns

```
SELECT * FROM a_table
```

- Example: select all the movie titles and their release year from the movies table

```
SELECT title, year FROM movies
```

SQL SELECT FROM

- Sometimes, you would want to rename a column for readability. You can rename a column in the query using AS keyword

- This is usually used when you think the original name is not good enough (e.g. too long, too abstract etc)

```
SELECT title AS movie_title FROM movies
```

SELECT DISTINCT

SQL SELECT DISTINCT

- The DISTINCT keyword can ensure the query output has unique row data
- Example: extract a unique list of first name from a name list

```
SELECT DISTINCT first_name FROM party_name_list
```

SQL SELECT DISTINCT

Table: party_name_list

first_name	last_name
Tom	Smith
Jerry	Jones
Lisa	Miller
Tom	Davis
Jerry	Johnson

```
SELECT
  first_name
FROM party_name_list
```

first_name
Tom
Jerry
Lisa
Tom
Jerry

```
SELECT DISTINCT
  first_name
FROM party_name_list
```

first_name
Tom
Jerry
Lisa

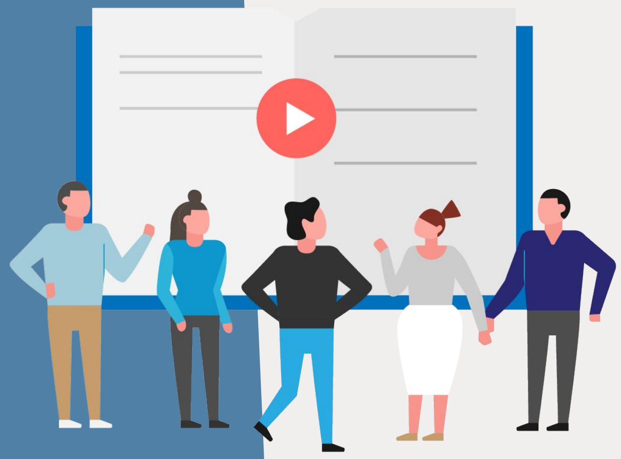
LIMIT and ORDER BY

</talentlabs>

SQL LIMIT

- When you look for some specific rows of the table instead of getting the full data, the LIMIT keyword can help to get the first few rows of the query result
- The number of rows can be set as any numbers you want

```
SELECT first_name FROM party_name_list LIMIT 3
```



SQL LIMIT

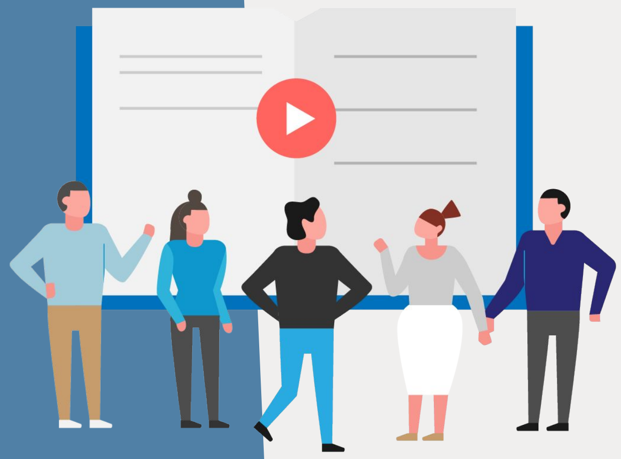
Table: party_name_list

first_name	last_name
Tom	Smith
Jerry	Jones
Lisa	Miller
Tom	Davis
Jerry	Johnson

```
SELECT
   first_name
FROM party_name_list
LIMIT 3
```

first_name
Jerry
Tom
Lisa

* Any 3 values from first_name will be returned as there is no specified sorting in the query



ORDER BY

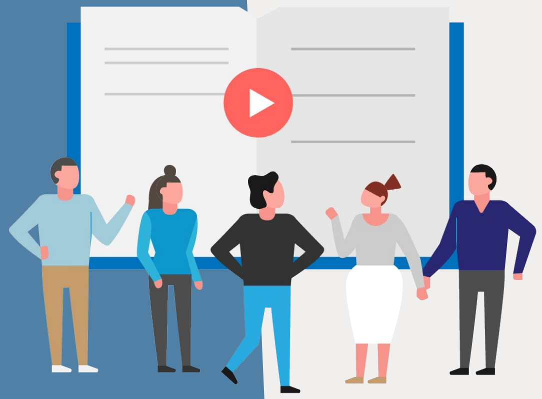
- Sometimes, we want to have a sorted results from database. For example, we want to results to be order by alphabetical order.
- You can use the ORDER BY keyword to specify columns which need to be sorted
- The sorting can be ascending (ASC) or descending (DESC), and can be multiple columns

```
SELECT first_name
FROM party_name_list
ORDER BY first_name ASC
```

Order by first name, in ascending order

```
SELECT first_name, last_name
FROM party_name_list
ORDER BY first_name, last_name DESC
```

Order by first name, then last name, in descending order



</talentlabs>

ORDER BY EXAMPLE

Table: party_name_list

first_name	**last_name**
Tom	Smith
Jerry	Jones
Lisa	Miller
Tom	Davis
Jerry	Johnson

```
SELECT
   first_name
FROM party_name_list
ORDER BY first_name ASC
```

first_name
Jerry
Jerry
Lisa
Tom
Tom

Ascending order

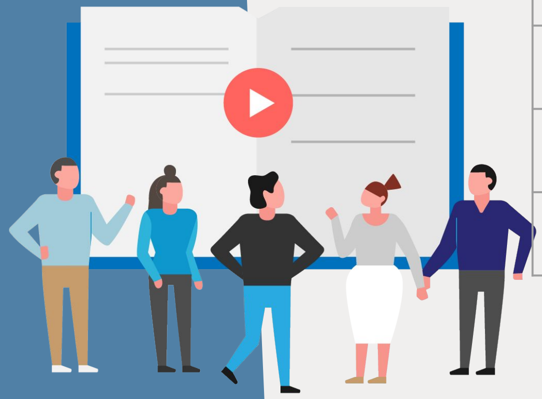
```
SELECT
   first_name, last_name
FROM party_name_list
ORDER BY first_name, last_name
DESC
```

first_name	**last_name**
Tom	Smith
Tom	Davis
Lisa	Miller
Jerry	Jones
Jerry	Johnson

Descending order

Combining LIMIT and ORDER BY

- As mentioned before, if you don’t include a ORDER BY block with LIMIT, then the database will only randomly return a few records to you.
- To make the results from LIMIT meaningful, you can combine LIMIT and ORDER BY
- E.g. ORDER BY exam_score LIMIT 3 will give your the top 3 students in the class



first_name	exam_score
Tom	80
Jerry	60
Lisa	90
Tom	50
Jerry	85

ORDER BY exam_score DESC LIMIT 3 →

first_name	exam_score
Lisa	90
Jerry	85
Tom	80

Combining LIMIT and ORDER BY

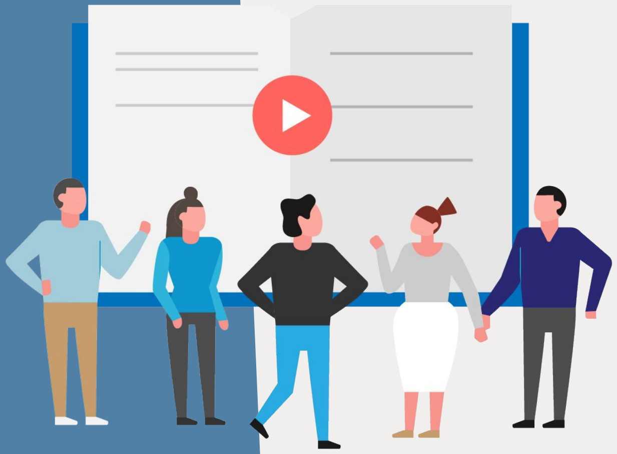
Table party_name_list

first_name	last_name
Tom	Smith
Jerry	Jones
Lisa	Miller
Tom	Davis
Jerry	Johnson

```
SELECT first_name
FROM party_name_list
ORDER BY first_name ASC
LIMIT 3
```

→

first_name
Jerry
Jerry
Lisa



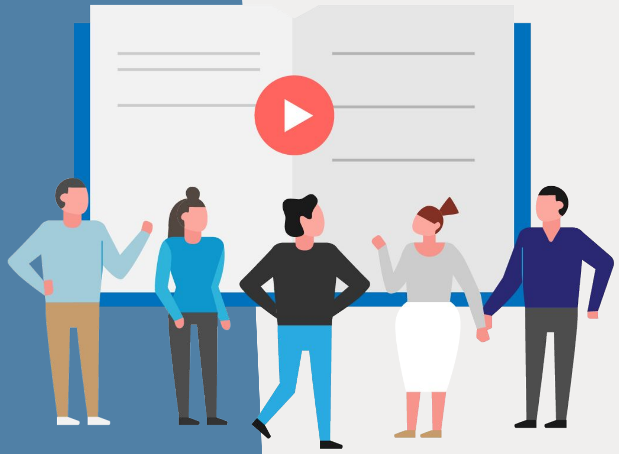
</talentlabs>

Summary

- We've learnt a few key SQL keywords

`SELECT FROM`	`LIMIT`
`SELECT DISTINCT`	`ORDER BY`

- Tried out the SQL keywords and query the movies database!



</talentlabs>

CHAPTER 5

SELECT Queries with Conditions

</talentlabs>

AGENDA

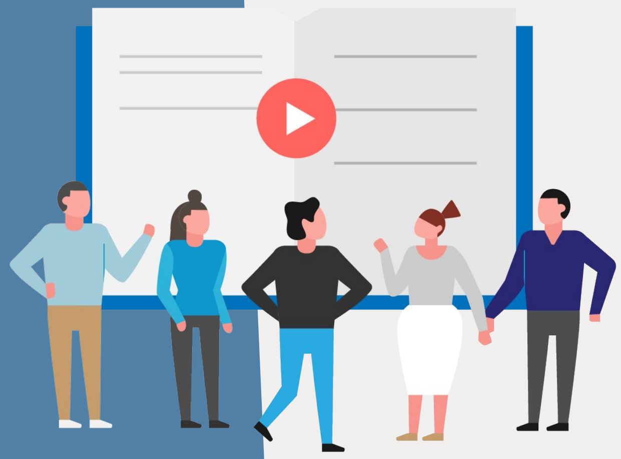
- WHERE clause
- Compound Conditions with AND/OR
- BETWEEN and IN
- LIKE

Where Clause

WHERE Clause

- WHERE clause is used to set a series of logic to filter out unwanted data
- We use different logic operators and SQL keywords to build conditions.

```
SELECT columns    FROM table    WHERE condition(s)
GROUP BY column(s)    ORDER BY column(s)    LIMIT n
```



WHERE Clause

- Using SELECT FROM, we query the whole movies table

```
SELECT * FROM movies
```

- If we want to specify to query movies of which year(s), we can add a comparison logic after a WHERE keyword

- Example: select all the movies with year equals 1990

```
SELECT * FROM movies WHERE year=1990
```

WHERE Clause

- WHERE clause supports common comparison operators

```
=   Equal
>   Bigger Than
<   Smaller Than
>=  Bigger Than or Equal
<=  Smaller Than or Equal
<>  Not equal
```

WHERE Clause for Strings

- Equal sign can also searches for ***exact match*** of the condition input to the column values
- To specify a text, use double quotes - ""

```
SELECT * FROM movies WHERE title = "The Lord of the Rings"
```

Matches

“The Lord of the Rings”

NOT Matching

“The Lord of the Rings: The Two Towers”
“The Lord of the Rings 2”
“The Lord of the Rings: Return of The King”

Compound Conditions

</talentlabs>

Compound Conditions - AND

Example 1: Get movies which release year is later than 1990 and earlier than 2000

```
SELECT * FROM movies
WHERE year > 1990 AND year < 2000
```

Compound Conditions - OR

Example 2: Get movies which title is The Lord of the Rings or Star Wars

```
SELECT * FROM movies
WHERE
   title = 'The Lord of the Rings'
   OR title = 'Star Wars'
```

Compound Conditions - NOT

- You can also use NOT keyword to represent “exception”

Example - movies that are not released after 2000

```
SELECT * FROM movies
WHERE
   NOT year > 2000
```

Compound Conditions - Parentheses

- As we are chaining more and more conditions and we want to make sure our logic is correct and clear, we can use Parentheses - ()

Example - movies that are not released between 1990 and 2000

```
Correct
SELECT * FROM movies
WHERE
   NOT (year > 1990 and year <2000)
```

</talentlabs>

Compound Conditions - Parentheses

- As we are chaining more and more conditions and we want to make sure our logic is correct and clear, we can use Parentheses - ()

Example - movies that are not released between 1990 and 2000

```
Correct
SELECT * FROM movies
WHERE
   NOT (year > 1990 AND year < 2000)
```

- movies that are before 1990
- moves that are after 2000

```
Wrong
SELECT * FROM movies
WHERE
   NOT year > 1990 AND year < 2000
```

ONLY movies that are before 1990

Simplifying Queries with BETWEEN and IN

</talentlabs>

BETWEEN

- We can use BETWEEN keyword to specific a range of value
- Consider the below SQL query

```
SELECT * FROM movies WHERE year >= 1995 AND year <=2010
```

- We can simplify the query like below

```
SELECT * FROM movies WHERE year BETWEEN 1995 AND 2010
```

- Note that 1995 and 2010 are included in the condition

IN

- We can use IN keyword to simplify a series of OR condition to a single field
- Consider the below SQL query

```
SELECT * FROM movies
WHERE
    title='The Lord of the Ring' OR title='Star Wars'
```

- We can simplify the query like below

```
SELECT * FROM movies
WHERE
    title IN ('The Lord of the Ring', 'Star Wars')
```

In-Class Exercise

- Rewrite this query to simplify it using IN and BETWEEN

```
SELECT * FROM movies
WHERE
    year=1995 OR year=1996 OR year=1997
    OR title='The Lord of the Ring' OR title='Star Wars'
```

In-Class Exercise

- Rewrite this query to simplify it using IN and BETWEEN

```
SELECT * FROM movies
WHERE
    year=1995 OR year=1996 OR year=1997
    OR title='The Lord of the Ring' OR title='Star Wars'
```

```
SELECT * FROM movies
WHERE
    year BETWEEN 1995 AND 1997
    OR title IN ('The Lord of the Ring', 'Star Wars')
```

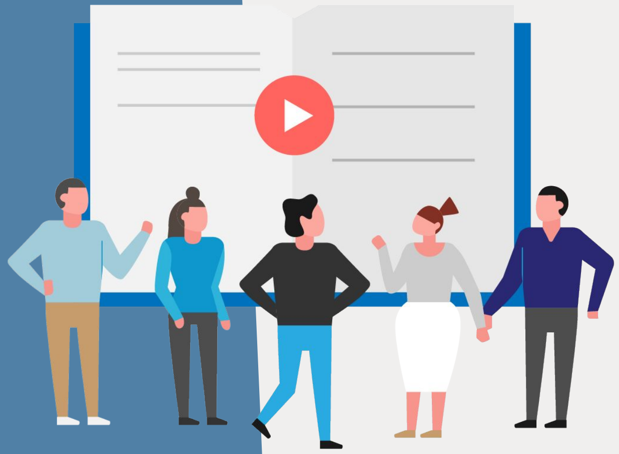
LIKE

LIKE

You can specify a text pattern to the condition using the LIKE keyword

Example 1: we want to select all movies which title is start with “Star Wars” we can query as below

```
SELECT * FROM movies
WHERE title LIKE 'Star Wars%'
```



SQL LIKE with _

Table: party_name_list

first_name	last_name
Tom	Smith
Jerry	Jones
Lisa	Miller
Tom	Davis
Jerry	Johnson

```
SELECT *
FROM party_name_list
WHERE
   last_name LIKE 'J____'
```

first_name	last_name
Jerry	Jones

- one low dash represents any single character. Here we have 4 i.e. 4 any characters.
- The word “Johnson” has more than 4 characters after “J”. So that row isn't returned

SQL LIKE with %

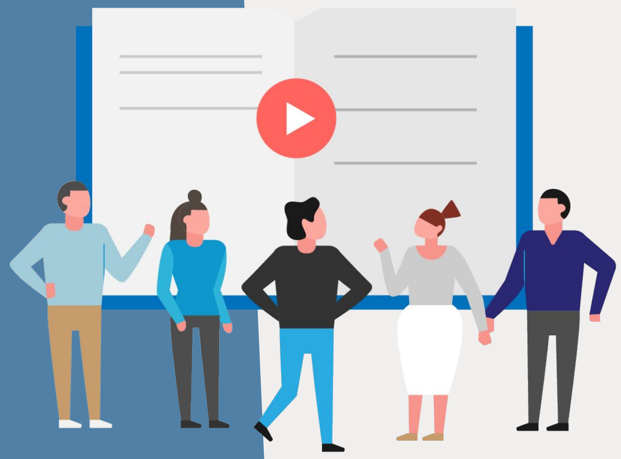
Table: party_name_list

first_name	last_name
Tom	Smith
Jerry	Jones
Lisa	Miller
Tom	Davis
Jerry	Johnson

```
SELECT *
FROM party_name_list
WHERE
    last_name LIKE 'J%'
```

- % represents any characters in any length (from 0 to any)

first_name	last_name
Jerry	Jones
Jerry	Johnson



SQL LIKE with %

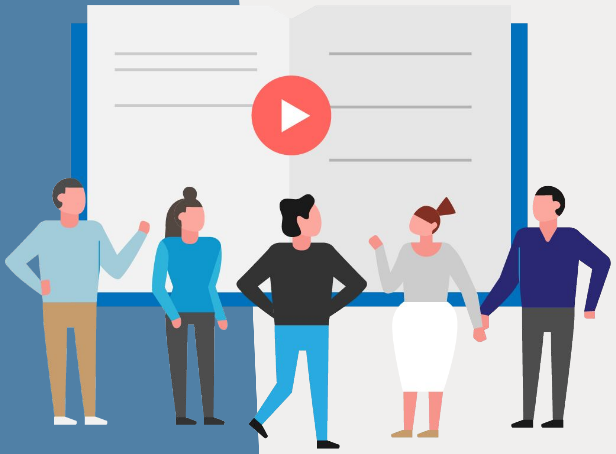
Table: party_name_list

first_name	**last_name**
Tom	Smith
Jerry	Jones
Lisa	Miller
Tom	Davis
Jerry	Johnson

```
SELECT *
FROM party_name_list
WHERE
    last_name LIKE '%i%'
```

- add % to both side of the keyword for search the keyword appearance in any position

first_name	**last_name**
Tom	Smith
Lisa	MIller
Tom	Davis



Summary

- We've learnt a few key SQL keywords for filtering data in the database

WHERE

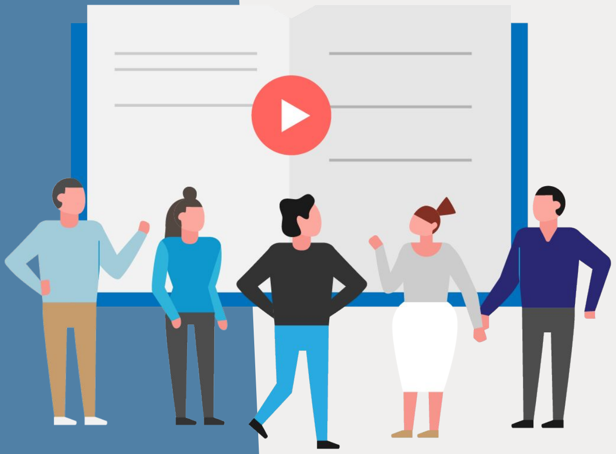
BETWEEN

AND

IN

LIKE

OR



</talentlabs>

CHAPTER 6

Complex SELECT Statements

</talentlabs>

AGENDA

- Subquery
- SQL CASE
- Basic Aggregations
- Advanced Aggregation
  - GROUP BY
  - HAVING

Subqueries

When queries getting complicated

- We may need to use the data of one table to query another table
- We may also need to filter or query a table twice to get the results we want
- Let’s assume we have the below table and columns

Movies

id	title
1	Movie A
2	Movie B
3	Movie C
4	Movie D
5	Movie E

Ratings

movie_id	rating
1	5.7
2	3.0
3	9.3
4	2.5
5	6.2

Table: movies

id	title
1	Toy Story
2	Toy Story 2
3	Toy Story 3
4	Star Wars 1
5	Star Wars 2

Table: ratings

movie_id	rating
1	5.7
2	3.0
3	9.3
4	2.5
5	6.2

Subquery

- From the sample tables, let's say we want to extract the movie rating of Toy Story

```
SELECT rating FROM ratings
WHERE movie_id = (
    SELECT id FROM movies WHERE title='Toy Story'
)
```

- We have a query to movies table for the movie id of "Toy Story" before we query the ratings table
- In this example, the query to movies table is a subquery
- The id data queried (id=1) from movies table is passed to the main query as a WHERE clause condition (movie_id=1)
- The result will be 5.7

Subquery

Table: movies

id	title
1	Toy Story
2	Toy Story 2
3	Toy Story 3
4	Star Wars 1
5	Star Wars 2

Table: ratings

movie_id	rating
1	5.7
2	3.0
3	9.3
4	2.5
5	6.2

Results

movie_id	rating
1	5.7
2	3.0
3	9.3

- If we need to get the ratings of Toy Story Series (i.e. all three episodes.)

```
SELECT movie_id, rating FROM ratings
WHERE movie_id IN (
    SELECT id
    FROM movies
    WHERE title LIKE 'Toy Story%'
)
```

- We can use IN keyword to pick up multiple result of the subquery

Table: years

movie_id	year
1	2000
2	2009
3	2013
4	1980
5	1983

Organizing Subqueries

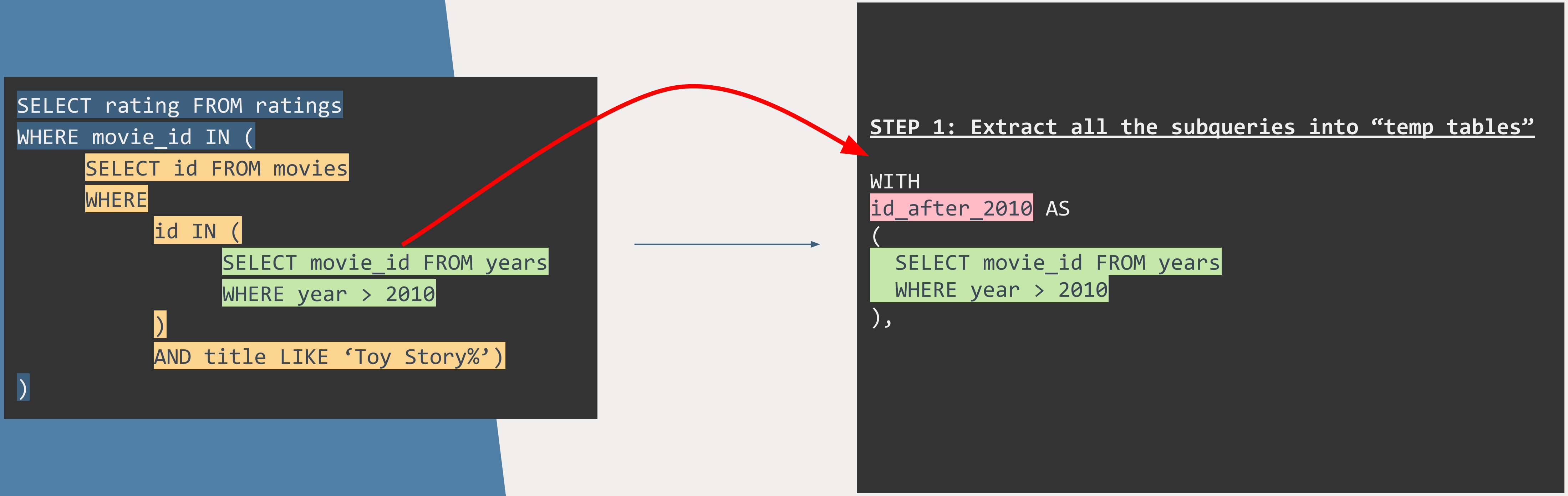
- Assume we have a years table for each movie_id
- Read the below query and try to tell what it is trying to accomplish.

```
SELECT rating FROM ratings
WHERE movie_id IN (
    SELECT id FROM movies
    WHERE
        id IN (
            SELECT movie_id FROM years
            WHERE year > 2010
        )
        AND title LIKE 'Toy Story%')
)
```

- The query is getting difficult to read as the subqueries are nested together

Organizing Subqueries

- We can use the WITH keyword to organize a long query especially when there are subqueries.
- WITH keyword enables you to customize subquery name to make the subqueries more meaningful.



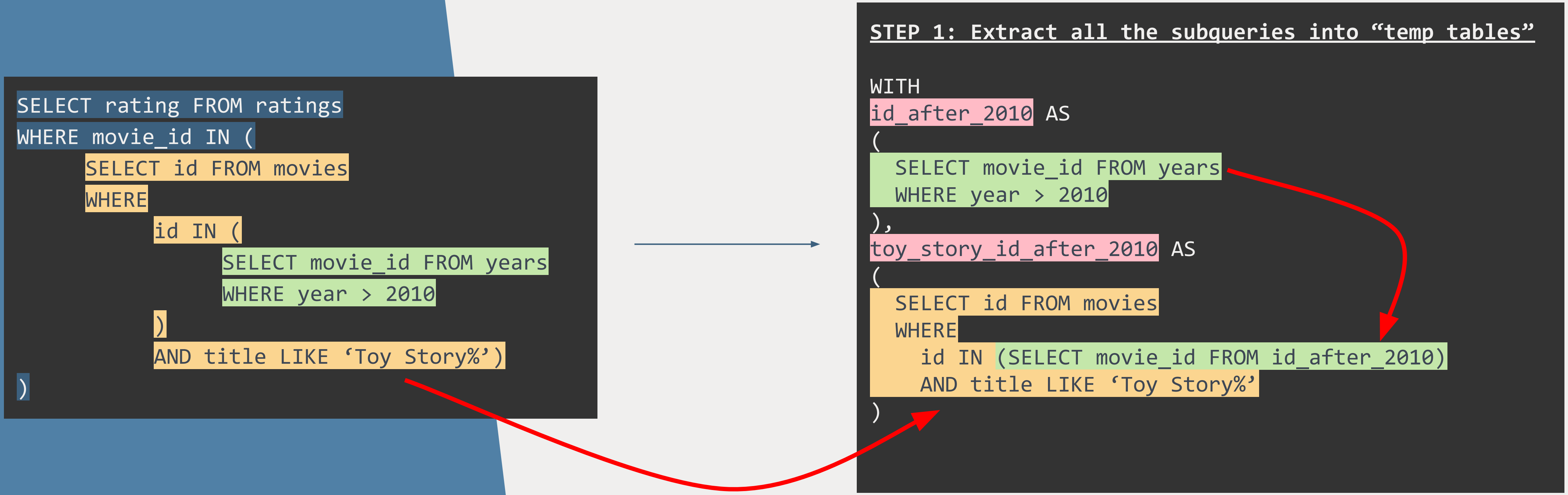
```
SELECT rating FROM ratings
WHERE movie_id IN (
      SELECT id FROM movies
      WHERE
            id IN (
                  SELECT movie_id FROM years
                  WHERE year > 2010
            )
            AND title LIKE ‘Toy Story%’)
)
```

STEP 1: Extract all the subqueries into “temp tables”

```
WITH
id_after_2010 AS
(
  SELECT movie_id FROM years
  WHERE year > 2010
),
```

Organizing Subqueries

- We can use the WITH keyword to organize a long query especially when there are subqueries.
- WITH keyword enables you to customize subquery name to make the subqueries more meaningful.

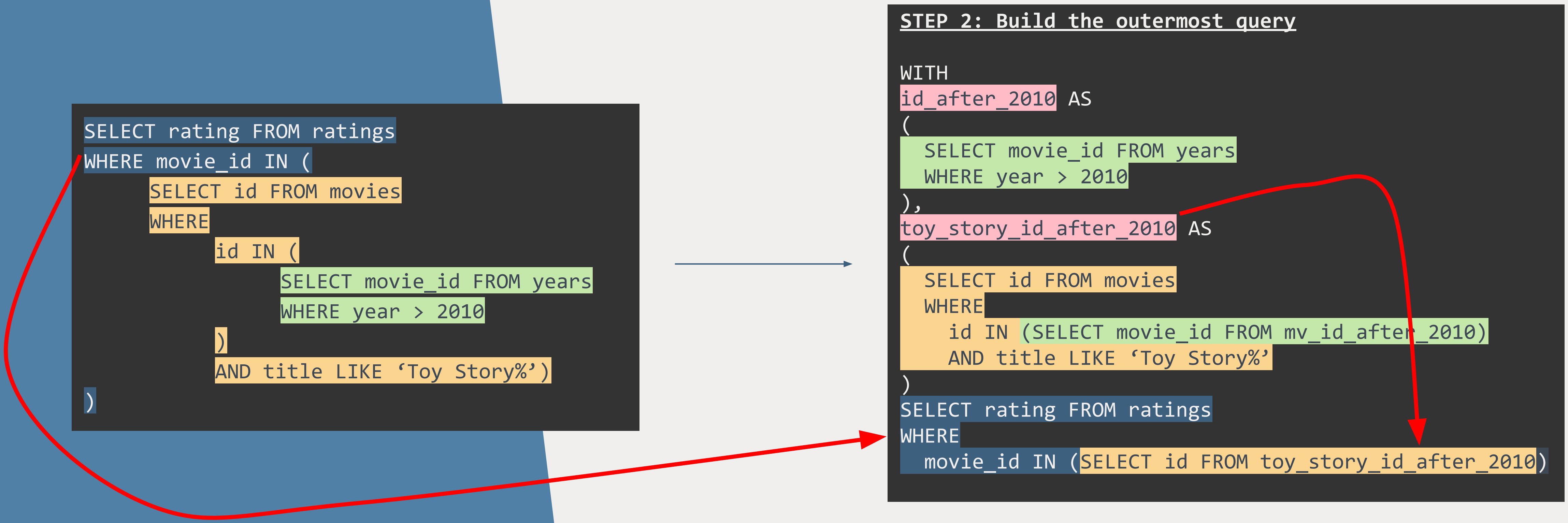


```
SELECT rating FROM ratings
WHERE movie_id IN (
      SELECT id FROM movies
      WHERE
            id IN (
                  SELECT movie_id FROM years
                  WHERE year > 2010
            )
            AND title LIKE ‘Toy Story%’)
)
```

```
STEP 1: Extract all the subqueries into “temp tables”

WITH
id_after_2010 AS
(
  SELECT movie_id FROM years
  WHERE year > 2010
),
toy_story_id_after_2010 AS
(
  SELECT id FROM movies
  WHERE
    id IN (SELECT movie_id FROM id_after_2010)
    AND title LIKE ‘Toy Story%’
)
```

Organizing Subqueries

- We can use the WITH keyword to organize a long query especially when there are subqueries.
- WITH keyword enables you to customize subquery name to make the subqueries more meaningful.



```
SELECT rating FROM ratings
WHERE movie_id IN (
      SELECT id FROM movies
      WHERE
            id IN (
                  SELECT movie_id FROM years
                  WHERE year > 2010
            )
            AND title LIKE ‘Toy Story%’)
)
```

STEP 2: Build the outermost query

```
WITH
id_after_2010 AS
(
  SELECT movie_id FROM years
  WHERE year > 2010
),
toy_story_id_after_2010 AS
(
  SELECT id FROM movies
  WHERE
    id IN (SELECT movie_id FROM mv_id_after_2010)
    AND title LIKE ‘Toy Story%’
)
SELECT rating FROM ratings
WHERE
  movie_id IN (SELECT id FROM toy_story_id_after_2010)
```

SQL CASE

Table: movies

id	title	year
1	Toy Story	2000
2	Toy Story 2	2009
3	Toy Story 3	2013
4	Star Wars 1	1980
5	Star Wars 2	1983

Query Result

title	movie_period
Toy Story	Released after 2000
Toy Story 2	Released after 2000
Toy Story 3	Released after 2000
Star Wars 1	Released before 2000
Star Wars 2	Released before 2000

SQL CASE

- CASE keyword can apply logic to manipulate values returned from a query
- It works like an IF-THEN-ELSE conditional statement of other programming languages

```
SELECT
    title,
    CASE
        WHEN year > 1999 THEN 'Released after 2000'
        ELSE 'Release before 2000'
    END AS movie_period
FROM movies
```

- Note 1: a CASE statement can include multiple conditions i.e. multiple WHEN-THEN.
- Note 2: ELSE clause is optional

Table: movies

id	title	year
1	Toy Story	2000
2	Toy Story 2	2009
3	Toy Story 3	2013
4	Star Wars 1	1980
5	Star Wars 2	1983

Query Result

title	movie_period
Toy Story	Released in 2000
Toy Story 2	Released in 2009
Toy Story 3	Released in 2013
Star Wars 1	Released in 1980
Star Wars 2	Released in 1983

SQL CASE

- We can also do value matching instead of just condition matching
- In this example, instead of condition matching (e.g. year>1999), we performs value matching on the “year” column

```
SELECT
    title,
    CASE year
        WHEN 2000 THEN 'Released in 2000'
        WHEN 2009 THEN 'Released in 2009'
        WHEN 2013 THEN 'Released in 2013'
        WHEN 1980 THEN 'Released in 1980'
        WHEN 1983 THEN 'Released in 1983'
    END AS movie_period
FROM movies
```

Basic Aggregation

Table: ratings

movie_id	rating
1	5.7
2	3.0
3	9.3
4	2.5
5	6.2

Basic Aggregations

- Sometimes, we might want to do some statistical analysis on the data (e.g. calculating sum, averages, maximum and minimum)
- This would help us in getting more insights about the data
- Say we want to know the average release year of the table. We can perform the following query using the AVG function

```
SELECT AVG(rating) FROM ratings
```

- The result would be 5.34

Basic Aggregations

Aggregation Function	Function
COUNT	counts how many rows are in a particular column
SUM	adds together all the values in a particular column
MIN and MAX	return the lowest and highest values in a particular column, respectively
AVG	calculates the average of a group of selected values

Table: ratings

movie_id	rating
1	5.7
2	3.0
3	9.3
4	2.5
5	6.2

COUNT

- Used to count number of records in the table

```
SELECT COUNT(*) FROM ratings
```

- Returns number of rows in the table, i.e. 5

Advanced Aggregation - GROUP BY

GROUP BY

Let’s consider the below data table:

Table: sample_movies

id	title	year	rating
1	A	1994	6.2
2	B	1994	7.2
3	C	1994	8
4	D	2009	6.2
5	E	2009	7.2
6	F	2009	9

How do we write **one** query to obtain the average movie rating of each year?

Table: sample_movies

id	title	year	rating
1	A	1994	6.2
2	B	1994	7.2
3	C	1994	8
4	D	2009	7.2
5	E	2009	8.2
6	F	2009	9

GROUP BY

- From what we’ve learnt, we can calculate the average of the rating by aggregation.
- Using WHERE clause, the rating data can be filtered by year

```
SELECT AVG(rating)
FROM sample_movies
WHERE year=1994
```

AVG(rating)
7.13

```
SELECT AVG(rating)
FROM sample_movies
WHERE year=2009
```

AVG(rating)
8.13

- However, it takes 2 queries instead of 1 to obtain the average rating. What if the table contains even more years?

GROUP BY

GROUP BY clause allows grouping of data by one or more fields and then perform aggregation by each grouping value

```
SELECT
    year,
    AVG(rating) AS avg_rating
FROM sample_movies
GROUP BY year
```

Table: sample_movies

id	title	year	rating
1	A	1994	6.2
2	B	1994	7.2
3	C	1994	8
4	D	2009	7.2
5	E	2009	8.2
6	F	2009	9

Results

year	avg_rating
1994	7.13
2009	8.13

Advanced Aggregation - HAVING

</talentlabs>

year	avg_rating
1994	7.13
2009	8.13

Filtering Aggregated Values

Scenario: Let's say we need to get the year and average rating which the average rating for the year is at least 8.

We cannot directly use WHERE clause to filter the aggregation results

```
SELECT
    year,
    AVG(rating) AS avg_rating
FROM sample_movies
WHERE AVG(rating)>=8 -- causes error
GROUP BY year
```

Table: sample_movies

id	title	year	rating
1	A	1994	6.2
2	B	1994	7.2
3	C	1994	8
4	D	2009	7.2
5	E	2009	8.2
6	F	2009	9

Subquery Results

year	avg_rating
1994	7.13
2009	8.13

Results

year	avg_rating
2009	8.13

Solution 1 - Using Subquery

We can use WHERE clause to filter aggregation results, but will need to leverage subquery to store the aggregation results first

```
SELECT
    year,
    avg_rating
FROM
(
    SELECT
        year,
        AVG(rating) AS avg_rating
    FROM sample_movies
    GROUP BY year
)
WHERE avg_rating >= 8 -- this works
```

Table: sample_movies

id	title	year	rating
1	A	1994	6.2
2	B	1994	7.2
3	C	1994	8
4	D	2009	7.2
5	E	2009	8.2
6	F	2009	9

Results

year	avg_rating
2009	8.13

Solution 2 - HAVING keyword

To simplify the query, we can use HAVING clause to filter aggregation result while keeping the query simple

Solution 2 with HAVING keyword
(Much longer query)

```
SELECT
    year,
    AVG(rating) AS avg_rating
FROM sample_movies
GROUP BY year
HAVING AVG(rating) >= 8
```

Solution 1 with Subquery
(Much longer query)

```
SELECT
    year,
    avg_rating
FROM
(
    SELECT
        year,
        AVG(rating) AS avg_rating
    FROM sample_movies
    GROUP BY year
)
WHERE avg_rating>=8
```

Summary

- We've learnt subqueries and WITH keyword for subquery organization
- We've learnt CASE statement for working with conditions for data values
- We've learnt aggregations, followed by GROUP BY and HAVING for data grouping and aggregation filtering

</talentlabs>

CHAPTER 7

Table Relationships

</talentlabs>

AGENDA

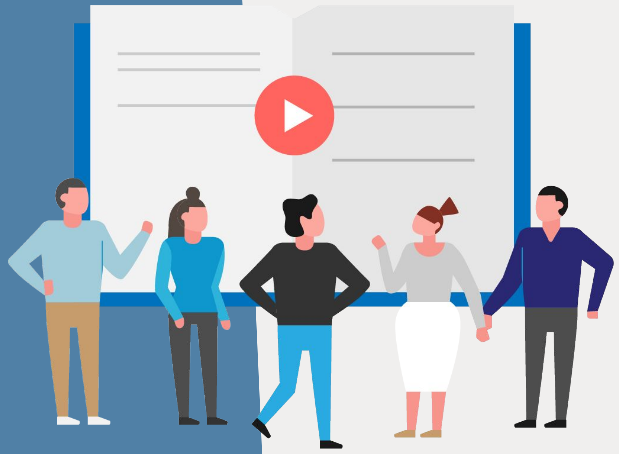
- Primary Key
- Foreign Key
- Relationships

Primary Key

</talentlabs>

Primary Key (PK)

- An ***unique*** identifier of the records, kind of like an ID
- e.g. For a city, citizens' names can be repeated. We add an id column to distinguish each citizen
- e.g. For an eCommerce Platform, purchases can be repeated. (same user purchases the same drink twice). So we'll add a transaction_id to each purchase record



Citizen

ID	First_name	Last_Name
1	Darren	Chiu
2	Peter	Chow
3	Michelle	Ling
4	Anthony	Chiu

Transactions

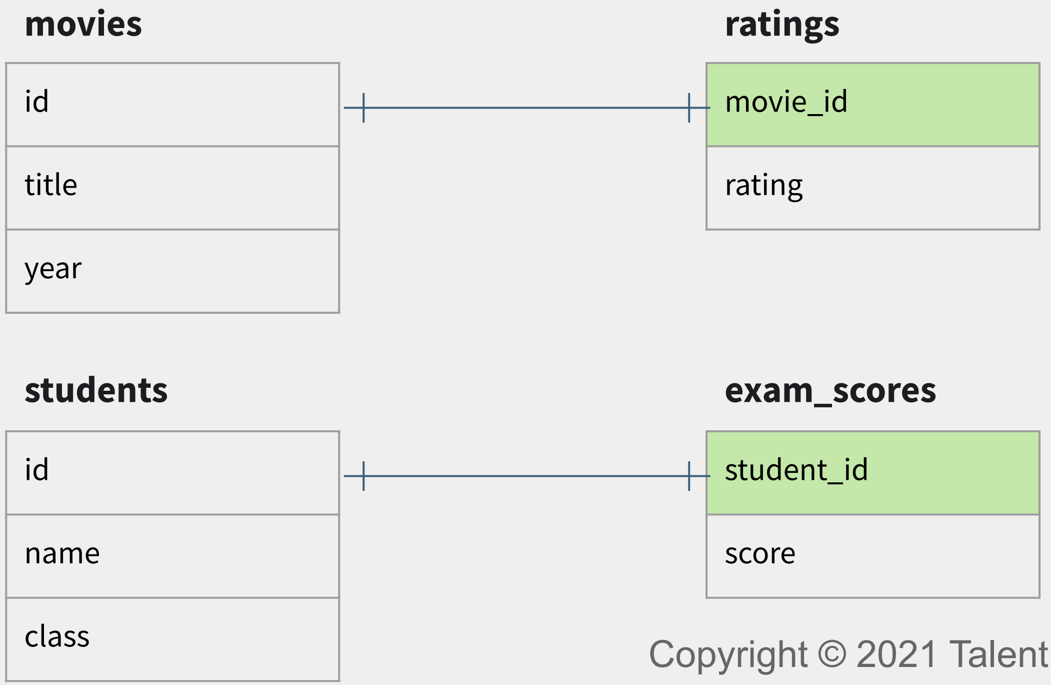
Transaction_ID	User	Product
1	Darren	Coke X 2
2	Peter	Burger X 2
3	Anthony	Chips
4	Darren	Coke X 2

</talentlabs>

Foreign Key

Foreign Key (FK)

- Foreign key is used to establish relationship between two tables
- e.g. For a movie ratings table, the movie_id column is created for identifying which movie the ratings is for. The movie_id column is “foreign key” column.



movies

id

title

year

ratings

movie_id

rating

students

id

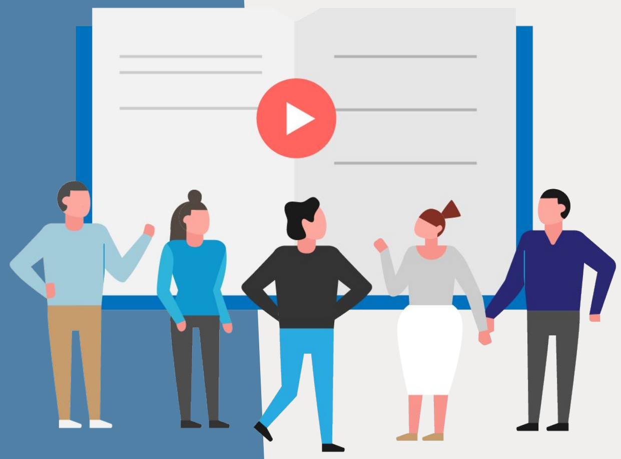
name

class

exam_scores

student_id

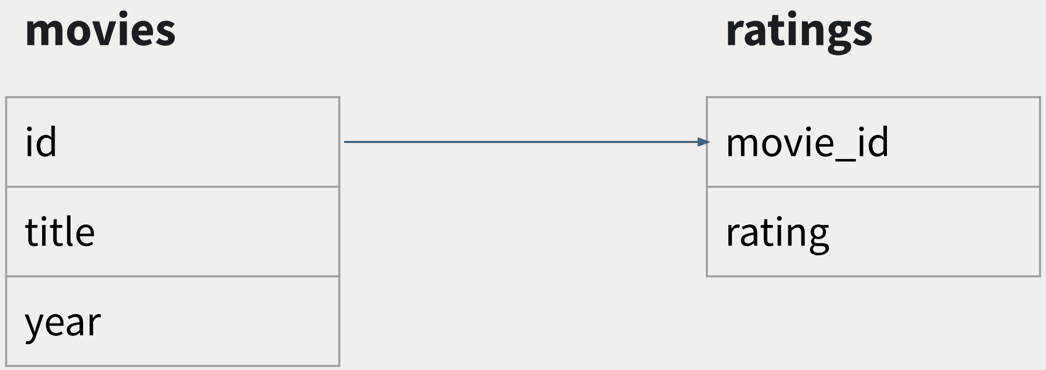
score



Relationships

What is Table Relationships

- We categorize data and organize the data into different tables
- For example, we put “movie details” in movies table, and the “movies ratings” into ratings table
- Different tables have relationships between them (e.g. each ratings is attached to a movie)



movies

id

title

year

ratings

movie_id

rating

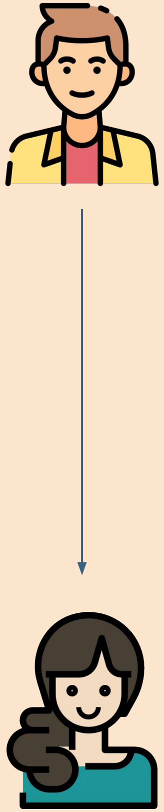
Three types of relationships

One-to-One relationship

One-to-Many relationship

Many-to-Many relationship

One-to-One Relationship



- One-to-One relationship are usually used to separate a big table into two, or attaching additional data to a record.
- E.g.
  - users and users_profile (separate a big users table to users and users_profile)
  - citizens and passport_info (attaching passport_info to citizen)

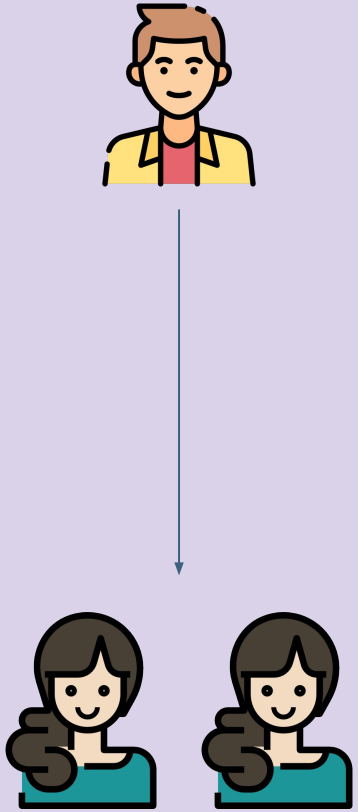
users

ID	First_name	Last_Name
1	Darren	Chiu
2	Peter	Chow
3	Michelle	Ling
4	Anthony	Chiu

users_profile

User_id	Gender	Country	Last_Online	Number of Posts
1	M	Hong Kong	2019-12-06	3
2	M	Japan	2019-12-05	2
3	F	Malaysia	2019-12-07	6
4	M	Hong Kong	2019-12-06	5

One-to-Many relationship



- One-to-Many relationship usually refers to ownership relationships
- E.g.
  - users and users_uploads (users owning multiple file uploaded)
  - country and states (country owning multiple states)

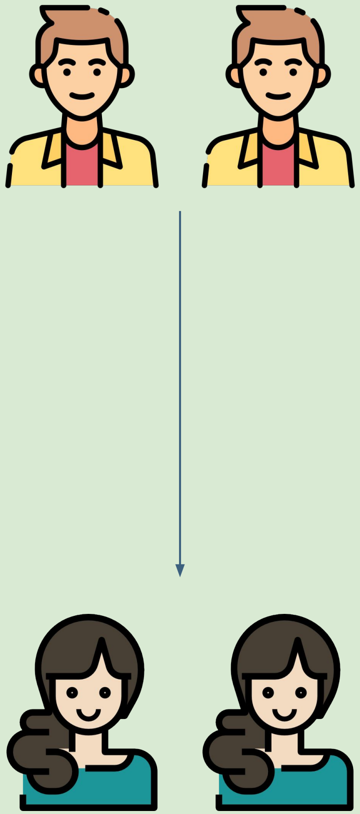
users

ID	First_name	Last_Name
1	Darren	Chiu
2	Peter	Chow
3	Michelle	Ling
4	Anthony	Chiu

users_uploads

User_id	Post_ID	Image	Text
1	1	img1.jpg	This is my first post.
2	2	img2.jpg	Hi Everyone!
3	3	img3.jpg	Please follow my page!
1	4	img4.jpg	Another post!
2	5	img5.jpg	I love posting!

Many-to-Many relationship



- Many-to-Many relationship usually refers to a membership between two categories of data
- E.g.
  - movies and actors (each actor is a “member” of the multiple movies)
  - desserts and ingredients (each ingredient is a “member of multiple dessert recipes)

Movies_Actors

movie_id	actor_id
1	1
1	3
1	4
2	1
2	2
3	1
3	4
4	1

Movies

id	movie_name
1	Toy Story
2	Star Wars
3	Harry Potter

Actors

id	name
1	Darren
2	Anthony
3	Peter
4	Karl

</talentlabs>

CHAPTER 8

SQL Joining

</talentlabs>

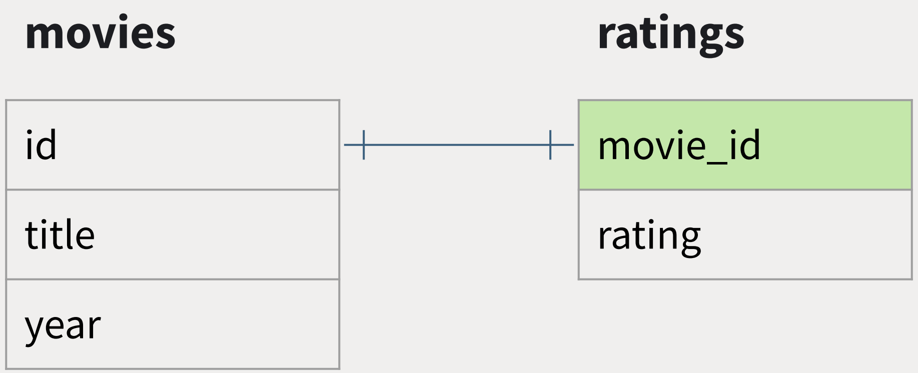
AGENDA

- What is Joining
- Types of Joins
    - Inner Join
    - Left Join & Right Join
    - Full Outer Join

What is Joining?

Joining

- Given there are relationship(s) between different tables, we can blend the two (or more) tables using JOIN keyword
- In general, joining can provide a more complete picture of a set of data e.g. movie data + rating data
- Let’s see how we write the query to link the tables together and return the result of the joined tables



movies

id

title

year

ratings

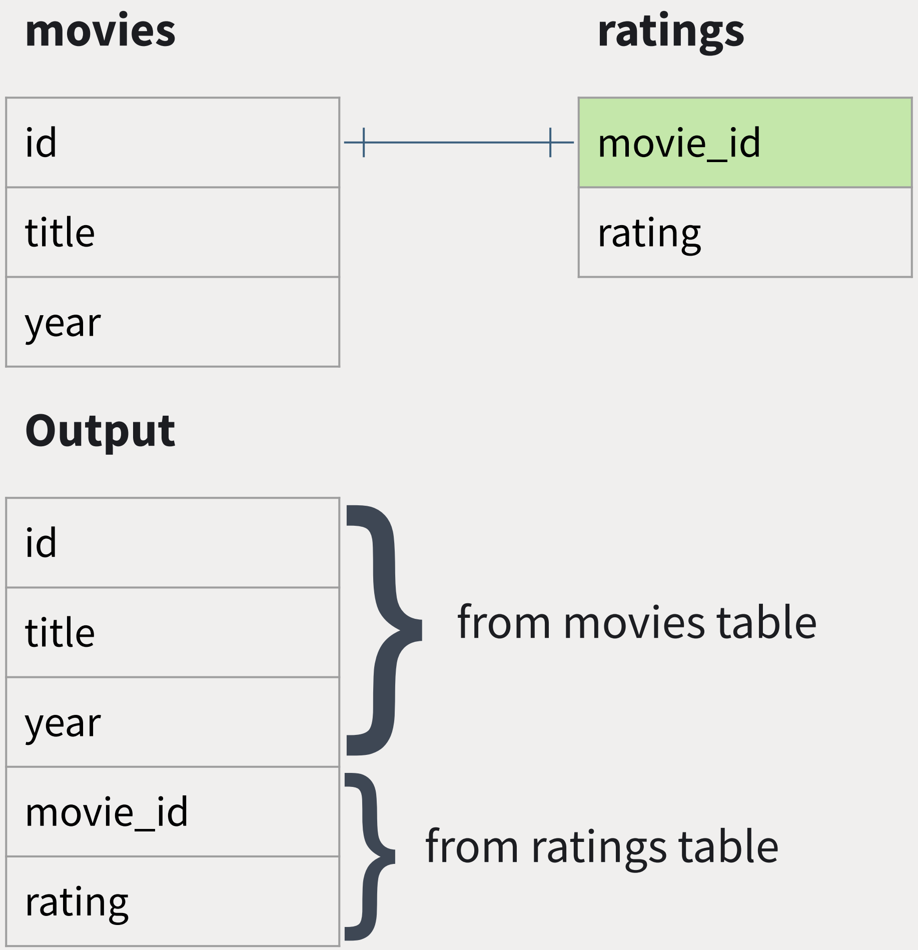
movie_id

rating

Quick Example on Joining

- Revisiting the relationship between movies and ratings, the primary key of movies table is “id”. The foreign key on ratings table is “movie_id”
- One-to-one relationships between movies and ratings

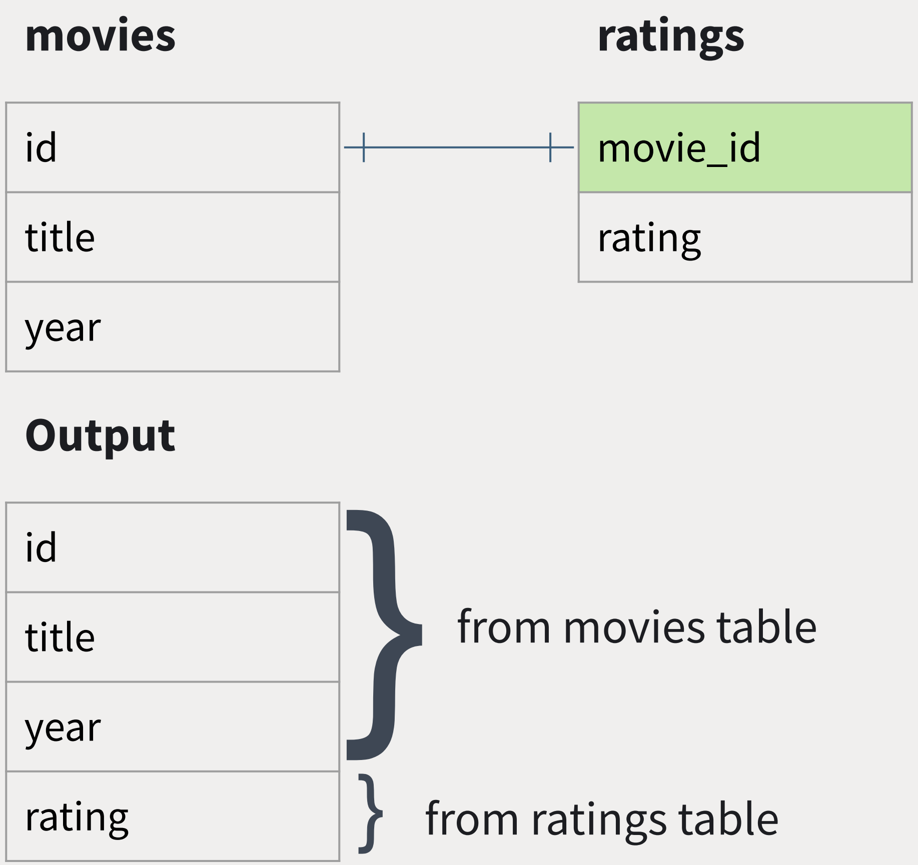
```
SELECT *
FROM
    movies JOIN ratings
        ON movies.id = ratings.movie_id
```



Quick Example on Joining

- Revisiting the relationship between movies and ratings, the primary key of movies table is “id”. The foreign key on ratings table is “movie_id”
- One-to-one relationships between movies and ratings

```
SELECT id, title, year, rating
FROM
    movies JOIN ratings
        ON movies.id = ratings.movie_id
```



movies

id

title

year

ratings

movie_id

rating

Output

id

title

year

rating

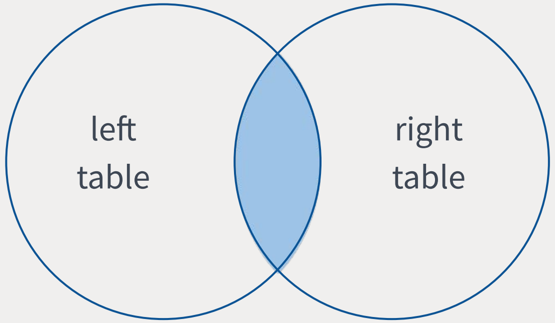
from movies table

from ratings table

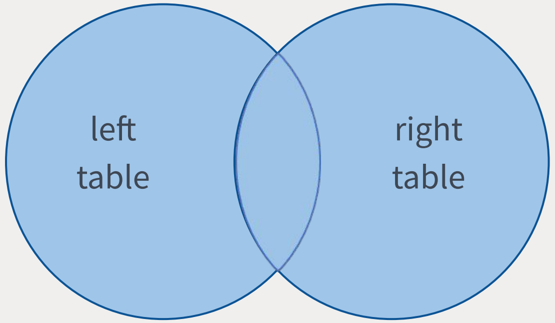
Types of Joining

Overview on Types of Joining

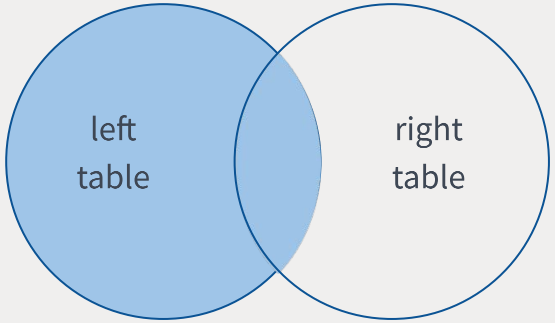
INNER JOIN



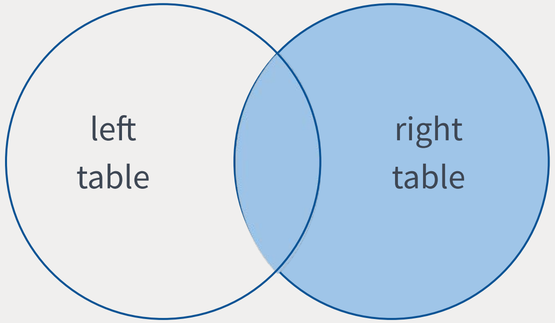
FULL JOIN / FULL OUTER JOIN



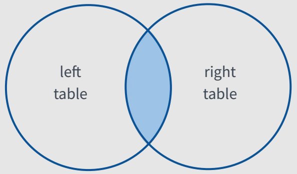
LEFT JOIN



RIGHT JOIN



Inner Join



Inner Join

movies

id	title
1	Star Wars
2	Harry Potter
3	Toy Story
4	Up

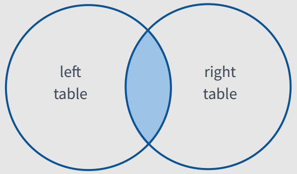
ratings

movie_id	rating
1	9.1
2	3.2
3	6.5
4	7.8

Joining Result

id	title	rating
1	Star Wars	9.1
2	Harry Potter	3.2
3	Toy Story	6.5
4	Up	7.8

```
SELECT id, title, rating
FROM
    movies JOIN ratings
        ON movies.id = ratings.movie_id
```



Inner Join

movies

id	title
1	Star Wars
2	Harry Potter
3	Toy Story
4	Up

ratings

movie_id	rating
2	3.2
3	6.5

Joining Result

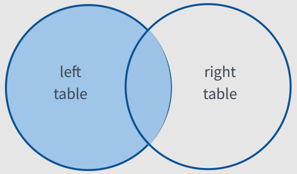
id	title	rating
2	Harry Potter	3.2
3	Toy Story	6.5

Note the joining result change when the **ratings** table (right table) has less data than **movies** table (left table)

```
SELECT id, title, rating
FROM
    movies JOIN ratings
        ON movies.id = ratings.movie_id
```

Left Join & Right Join

</talentlabs>



Left Join

movies

id	title
1	Star Wars
2	Harry Potter
3	Toy Story
4	Up

ratings

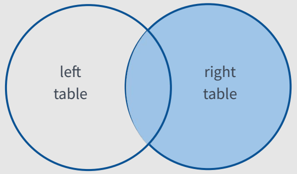
movie_id	rating
2	3.2
4	7.8

Joining Result

id	title	rating
1	Star Wars	
2	Harry Potter	3.2
3	Toy Story	
4	Up	7.8

```
SELECT id, title, rating
FROM
    movies LEFT JOIN ratings
        ON movies.id = ratings.movie_id
```

Not Available in SQLite



Right Join

movies

id	title
1	Star Wars
2	Harry Potter
3	Toy Story
4	Up

ratings

movie_id	rating
2	3.2
4	7.8
5	1.1

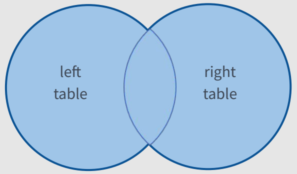
Joining Result

id	title	movie_id	rating
2	Harry Potter	2	3.2
4	Up	4	7.8
		5	1.1

```
SELECT id, title, movie_id, rating
FROM
    movies RIGHT JOIN ratings
        ON movies.id = ratings.movie_id
```

Full Outer Join

Not Available in SQLite



Full Outer Join

movies

id	title
1	Star Wars
2	Harry Potter
3	Toy Story
4	Up

ratings

movie_id	rating
2	3.2
4	7.8
5	1.1

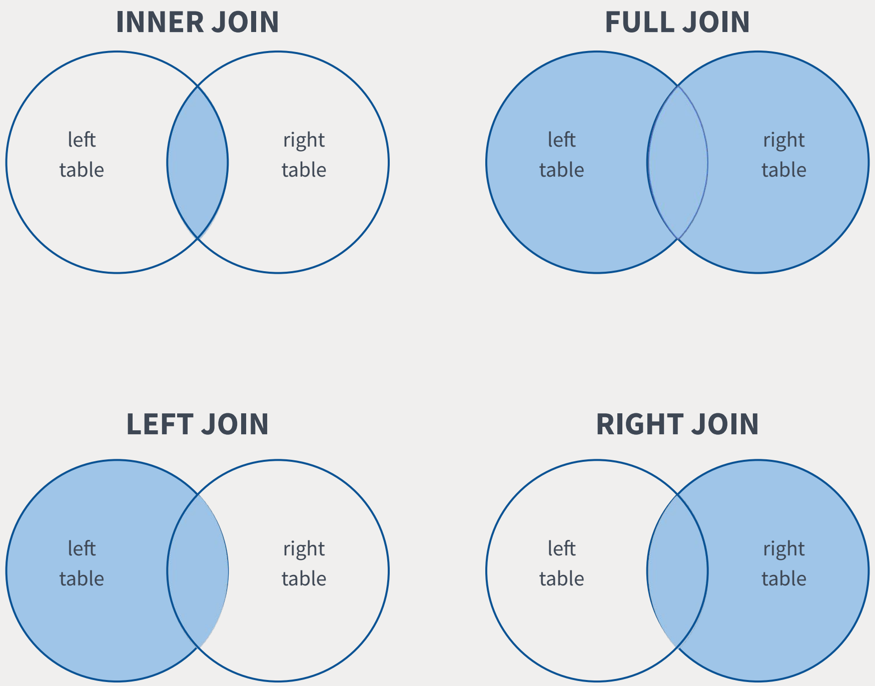
Joining Result

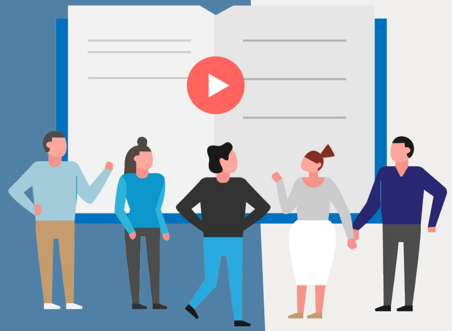
id	title	movie_id	rating
1	Star Wars		
2	Harry Potter	2	3.2
3	Toy Story		
4	Up	4	7.8
		5	1.1

```
SELECT id, title, movie_id, rating
FROM
    movies FULL OUTER JOIN ratings
        ON movies.id = ratings.movie_id
```

Summary

- We've learnt joining can blend data on multiple tables
- We've learnt different kinds of joining
  - Inner Join
  - Left Join
  - Right Join
  - Full Outer Join





</talentlabs>

8.6 Reading

Querying Tables with Relationships

Study Instructions

- In this reading materials, we have provided example code for querying tables with relationships for your reference. There are no new SQL concepts or SQL keywords. These examples are just leveraging SQL knowledge that you already know.

- Please read through the examples to make sure you understand each line of code. You will be using these techniques in your lab assignments.

</talentlabs>

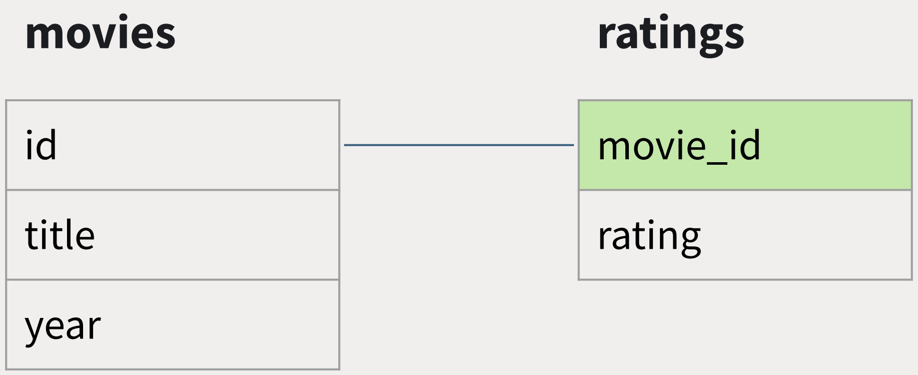
AGENDA

- Principles
- One-to-one relationships
- One-to-many relationships
- Many-to-many relationships
- Conclusions

Principles

Querying Tables with Relationships

- When you want to query tables with relationships, it’s all about joining.
- There are no special techniques, all you need to do is to identify the key relationships between tables (i.e. what is the foreign key linking the two tables)
- After identifying the key, then you just need to join the relevant tables together with the foreign keys.



movies

id
title
year

ratings

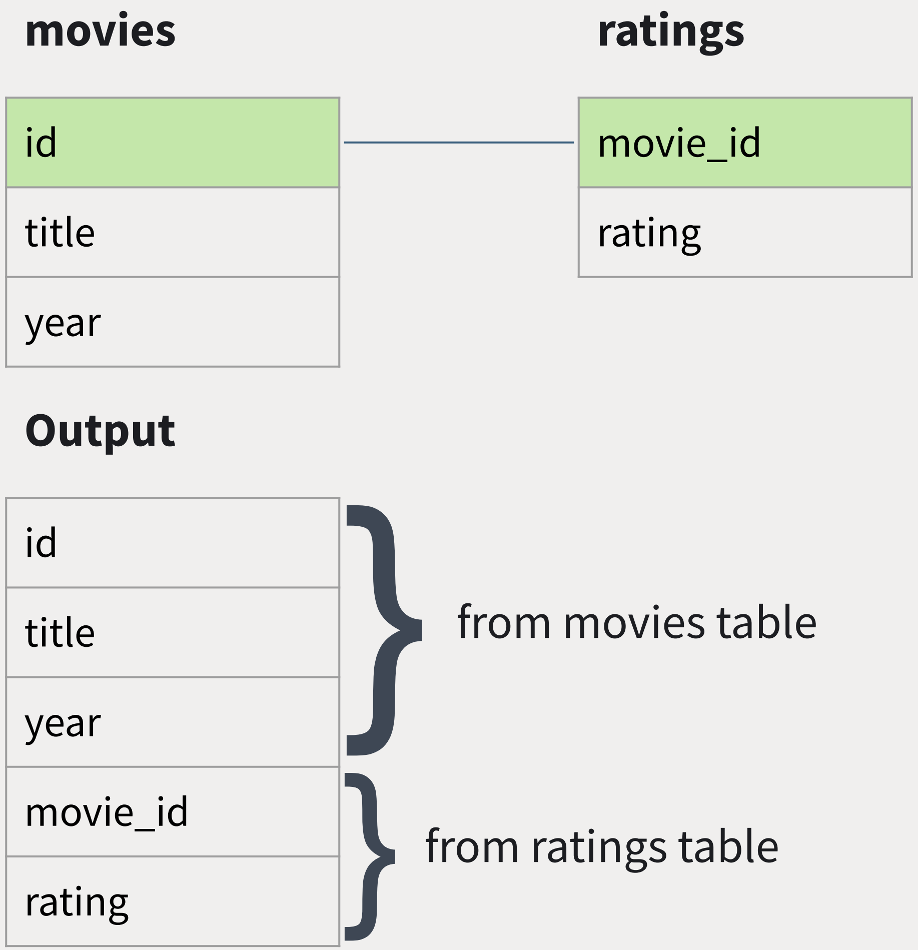
movie_id
rating

One-to-one Relationships

Joining One-to-one Relationships

- Movies and Ratings are in one-to-one relationship
- The key linkage between them is movies.id and ratings.movie_id (Foreign Key)
- So we only need to join the 2 tables on these two keys

```
SELECT *
FROM
    movies JOIN ratings
        ON movies.id = ratings.movie_id
```



movies

id
title
year

ratings

movie_id
rating

Output

id
title
year
movie_id
rating

from movies table

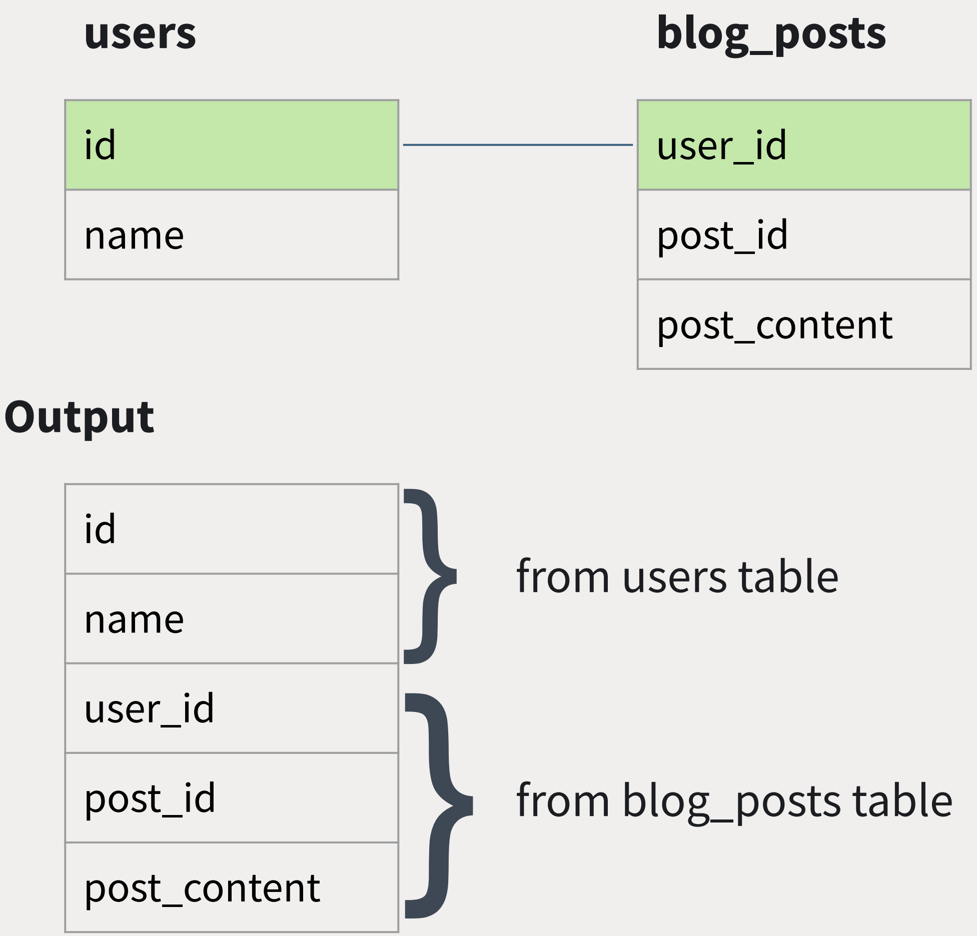
from ratings table

One-to-many Relationships

Joining One-to-Many Relationships

- Let’s say you are running a blogging website
- Users table and blog_posts table are in one-to-many relationships, i.e. each users can have multiple blog_posts
- For key relationships, the blog_posts.user_id (foreign key) is linked to users.id
- When you are joining the tables, it’s similar to one-to-one relationship, you only need to join on the foreign keys

```
SELECT *
FROM
    blog_posts join users
        ON blog_posts.user_id = users.id
```



users

id
name

blog_posts

user_id
post_id
post_content

Output

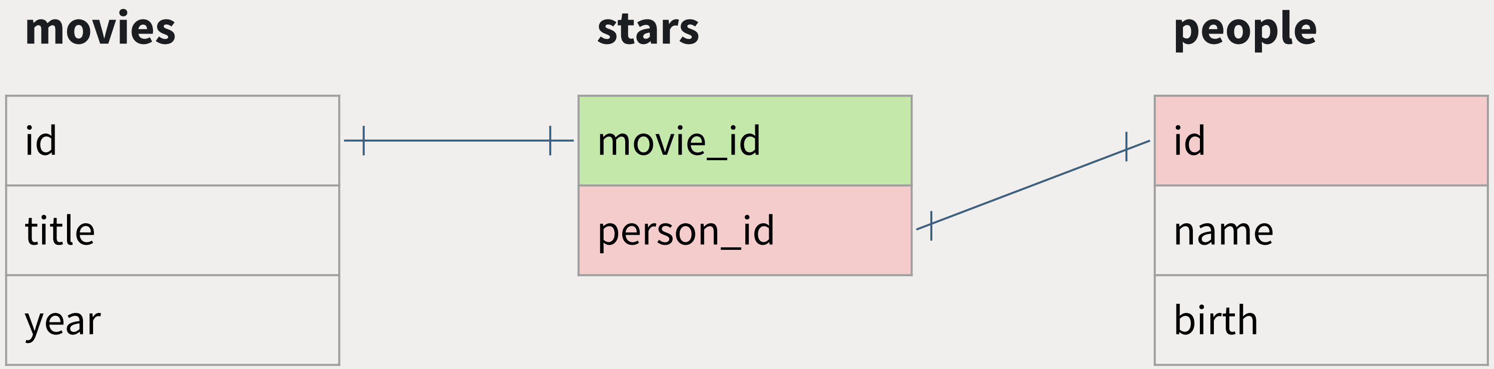
id
name
user_id
post_id
post_content

Many-to-many Relationships

</talentlabs>

Many-to-Many Relationships

- The relationship between “movies - stars - people” is a many-to-many relationship
- Each movie would have multiple actors, and each person would be acting in multiple movies
- There is an intermediate “relationship” table to store the mapping between movies and people (the “stars” table)



movies

- id
- title
- year

stars

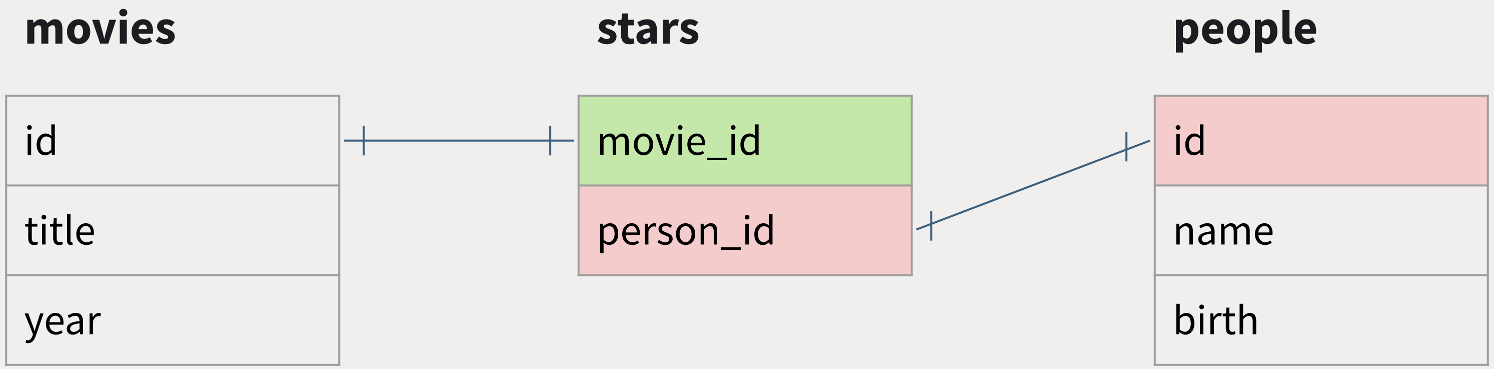
- movie_id
- person_id

people

- id
- name
- birth

Challenge of Joining Many-to-Many Relationships

- SQL Joins allows joining 2 tables at a time only
- In many-to-many relationships, there are 3 tables, so we need to do 2 joins
- We will first join the "movies with stars", then join the output with the "people" table



movies

- id
- title
- year

stars

- movie_id
- person_id

people

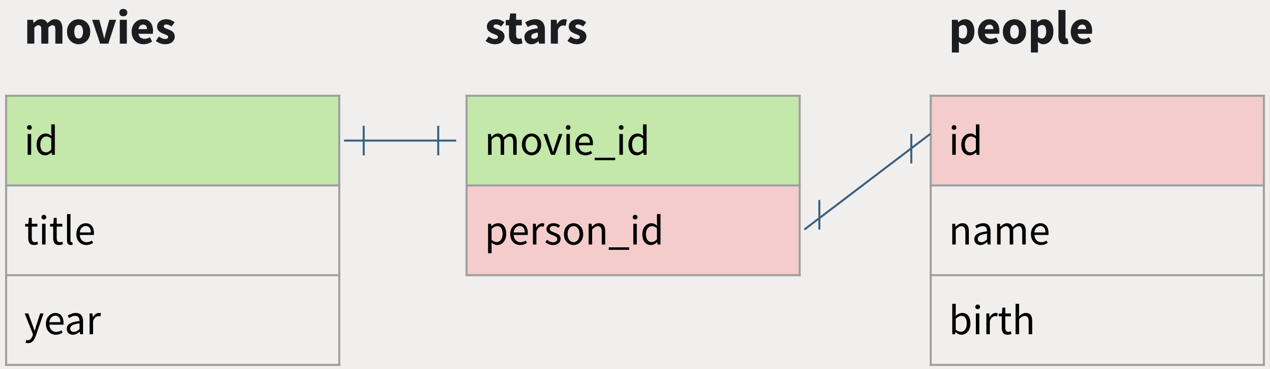
- id
- name
- birth

Joining Many-to-Many Relationships

First Join

Second Join

```
SELECT *
FROM
    (movies join stars
        ON movies.id = stars.movie_id) a
         join people on people.id = a.person_id
```

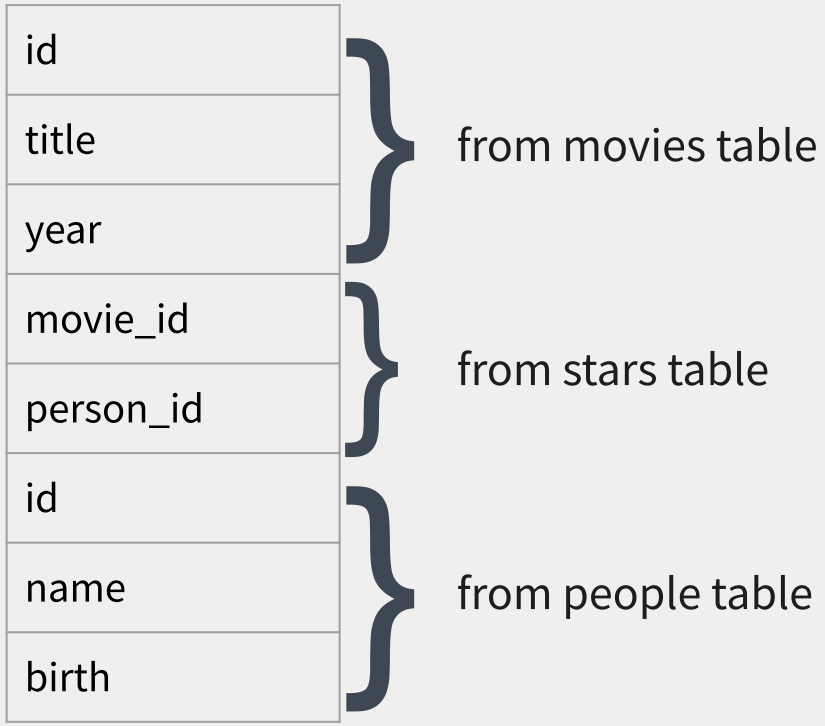


movies
id
title
year

stars
movie_id
person_id

people
id
name
birth

Output



id
title
year
movie_id
person_id
id
name
birth

</talentlabs>

Conclusions

</talentlabs>

Conclusions

- Table relationships actually **do not matter** a lot when you are pulling the data. The only important thing the key linkages.

- All you need to do is to identify tables that contains your data (2 tables or 3 tables), and identify the foreign key linkage between the tables. Then you can just use "join" to combine multiple tables together.

</talentlabs>

CHAPTER 9

Other SQL operations

</talentlabs>

AGENDA

- UNION / UNION ALL
- CREATE, ALTER and DROP Table
- INSERT, UPDATE and DELETE Records

UNION / UNION ALL

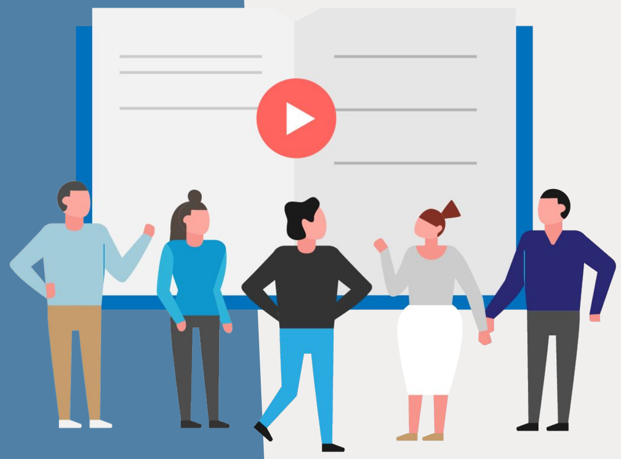
</talentlabs>

UNION

- UNION and UNION ALL are SQL keywords to glue data on multiple tables together vertically
- UNION will return **unique rows (distinct)** on the combined table
- UNION ALL will return **all the row data regardless of duplications**

```
SELECT year
FROM movies
UNION
SELECT year
FROM tv_shows
```

```
SELECT year
FROM movies
UNION ALL
SELECT year
FROM tv_shows
```



UNION / UNION ALL Examples

movies

year	title
1978	Star Wars
2004	Harry Potter
1998	Toy Story
2005	Up

tv_shows

year	title
2002	The Wire
2004	House
2008	Breaking Bad
1998	Cowboy Bebop

```
SELECT year
FROM movies
UNION
SELECT year
FROM tv_shows
```

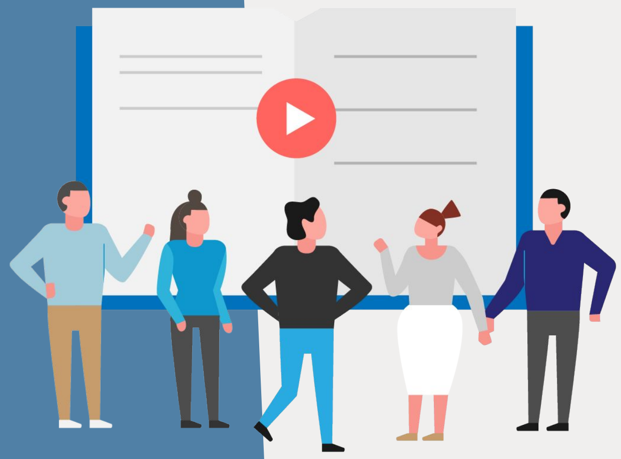
UNION result

year
1978
2004
1998
2005
2002
2008

```
SELECT year
FROM movies
UNION ALL
SELECT year
FROM tv_shows
```

UNION ALL result

year
1978
2004
1998
2005
2002
2004
2008
1998



</talentlabs>

CREATE, ALTER and DROP Table

Create table

- Provide table name, column name, and data type of the column
- Syntax:

```
CREATE TABLE table_name (
    column_name_1 data_type,
    column_name_2 data_type,
    ...
)
```

- Example:

```
CREATE TABLE movies (
    id INTEGER,
    name TEXT,
    year INTEGER
)
```

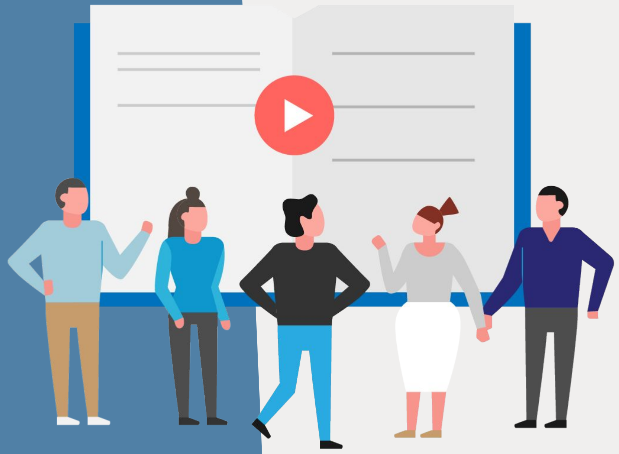
movies

id	name	year
xxx	xxx	xxx
…		
…		
…		

ALTER Table

- Provide table name, the operation to the table
- Common syntax examples

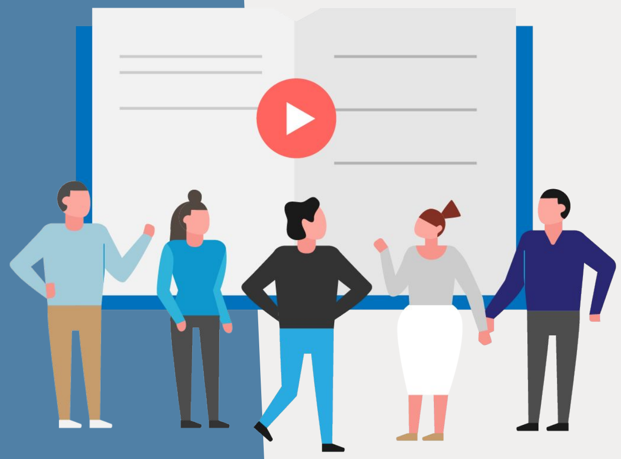
```
ALTER TABLE table_name
ADD column_name datatype;

ALTER TABLE table_name
DROP COLUMN column_name;

ALTER TABLE table_name
RENAME COLUMN current_name TO new_name;

ALTER TABLE table_name
RENAME TO new_table_name;
```



DROP Table

- Provide table name to be dropped
- ***** use with caution *** as you will be delete the whole table along with the data in it.**
- Example

```
DROP TABLE movies
```



Insert, Update and Delete Records

</talentlabs>

Insert records into a table

- Provide table name, column name, and data values of the column
- Syntax:

```
INSERT INTO table_name
(column_name_1, column_name_2)
VALUES
(data_value_1, data_value_2)
```

- Example:

```
INSERT INTO movies
(id, name, year )
VALUES
(1, ‘Toy Story’, 2005)
```

movies

id	name	year
1	Toy Story	2005
…		
…		
…		

Update records

- Provide table name, column name, data values and update condition

```
UPDATE table_name
SET column1 = value1, column2 = value2 …
where condition
```

```
UPDATE movies
SET name = Star Wars, year = 1978
where id = 1
```

UPDATE Example

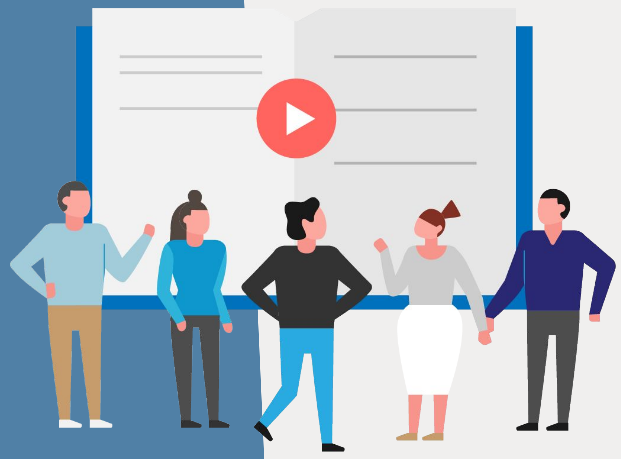
movies

year	title
	Star Wars
2004	Harry Potter
1998	Toy Story
	Up

```
UPDATE movies
SET year = 0000
where year is null
```

movies

year	title
0000	Star Wars
2004	Harry Potter
1998	Toy Story
0000	Up



</talentlabs>

Delete records

- Provide table name, delete condition
- Syntax:

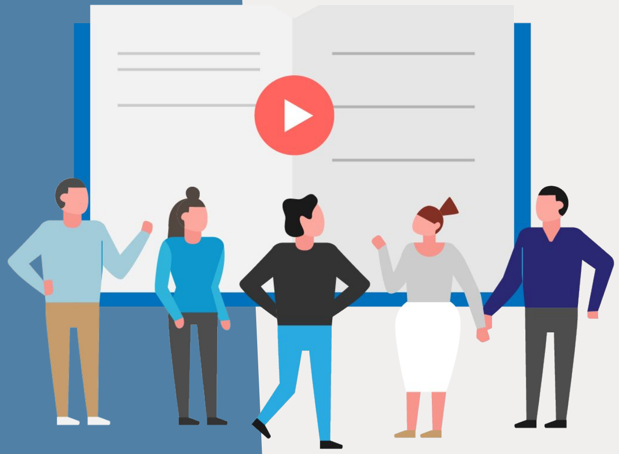
```
DELETE FROM table_name WHERE condition
```

- Example:

```
DELETE FROM movies where id = 1
```

Summary

- We've learnt various additional operation of SQL
  - Combining Tables
    - UNION
    - UNION ALL
  - Making Changes to Tables
    - CREATE TABLE
    - ALTER TABLE
    - DROP TABLE
  - Making Changes to Records
    - INSERT INTO
    - UPDATE
    - DELETE



</talentlabs>

</talentlabs>

CHAPTER 10

ER Diagram

</talentlabs>

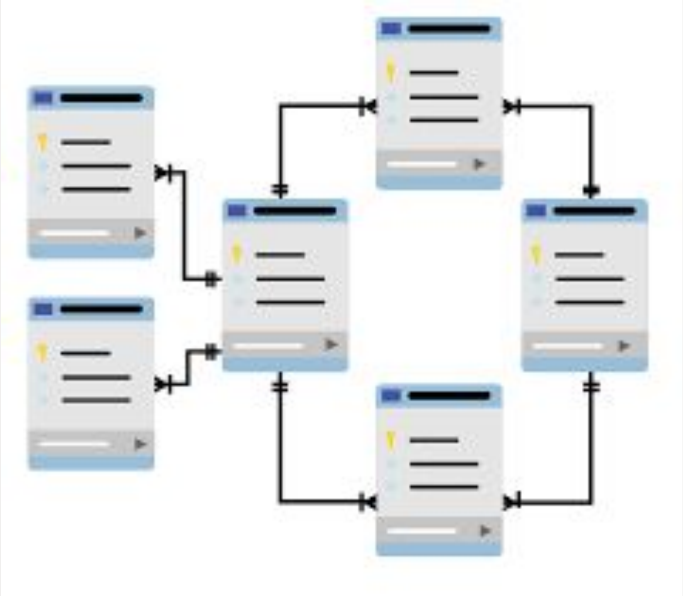
AGENDA

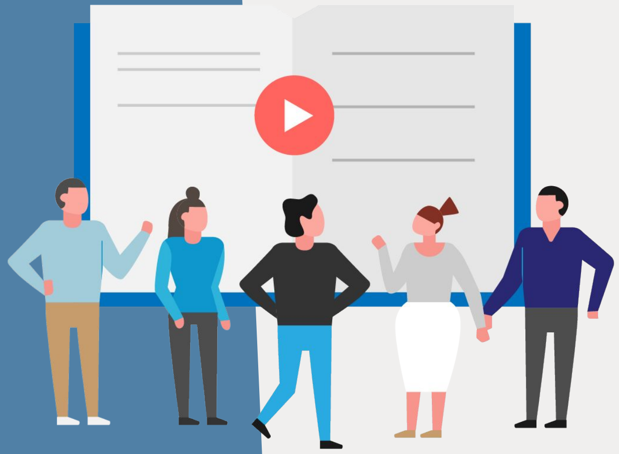
- ER Diagram
- One-to-Many Relationship
- One-to-One Relationship
- Many-to-Many Relationship

ER Diagram

ER Diagram (ERD)

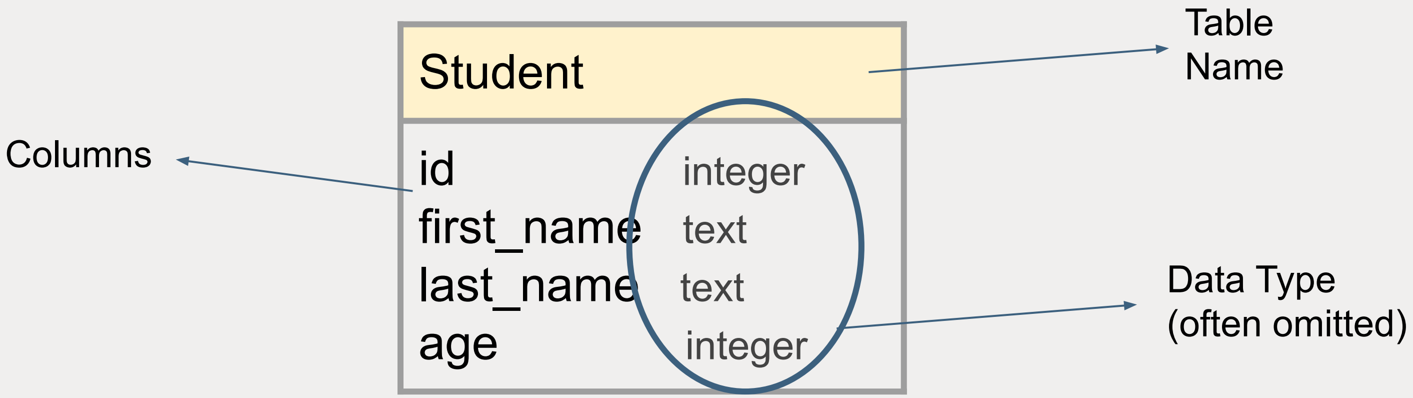
- Entity Relationship Diagram
- Usually for formal design of database
- For anyone who are new to a database can quickly understand the database architecture





</talentlabs>

Basic Module of ER Diagram - Table

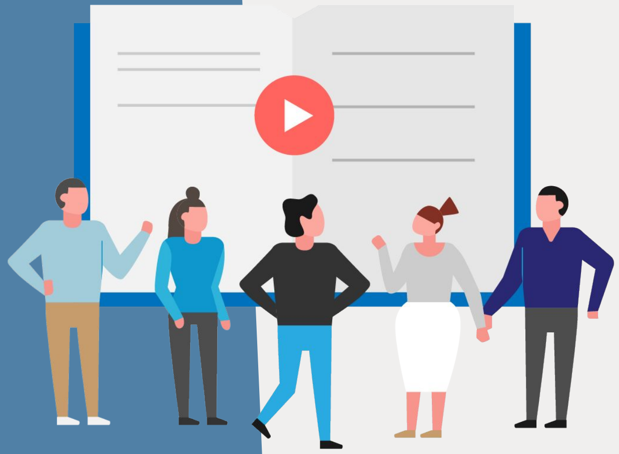


Student	
id	integer
first_name	text
last_name	text
age	integer

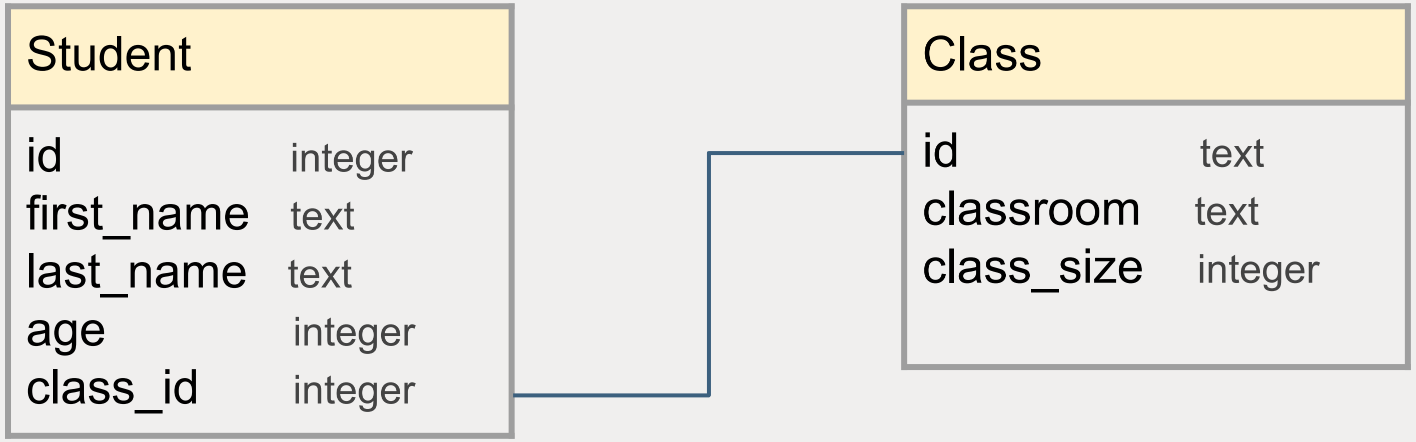
Table Name

Columns

Data Type
(often omitted)

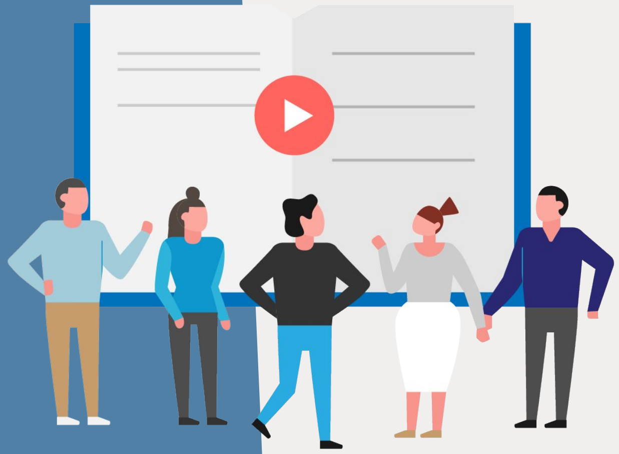


Representing Relationship



Student	
id	integer
first_name	text
last_name	text
age	integer
class_id	integer

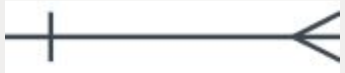
Class	
id	text
classroom	text
class_size	integer



One-to-Many Relationship

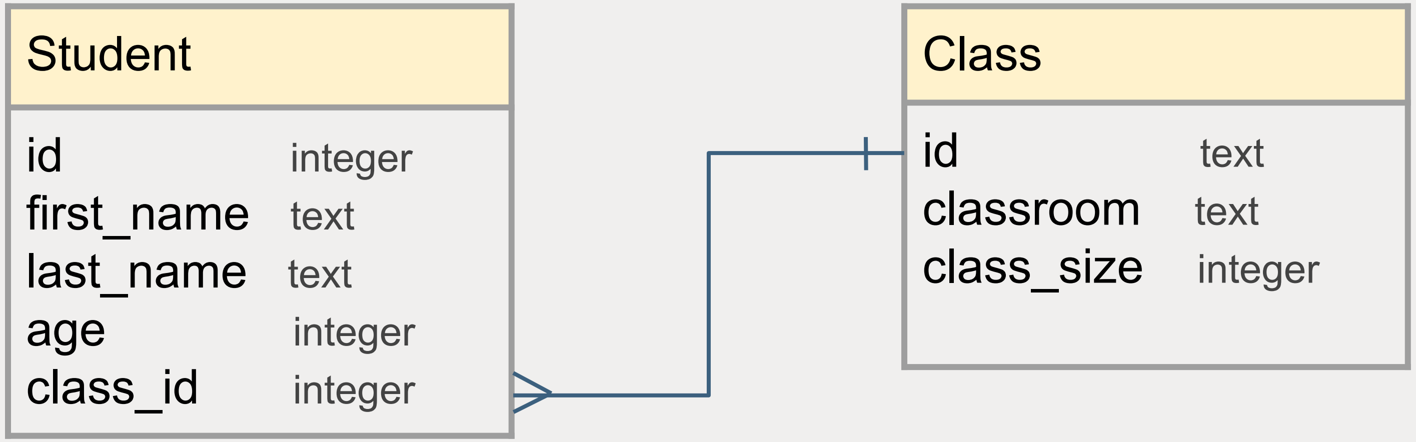
One-to-Many Relationship

One Side



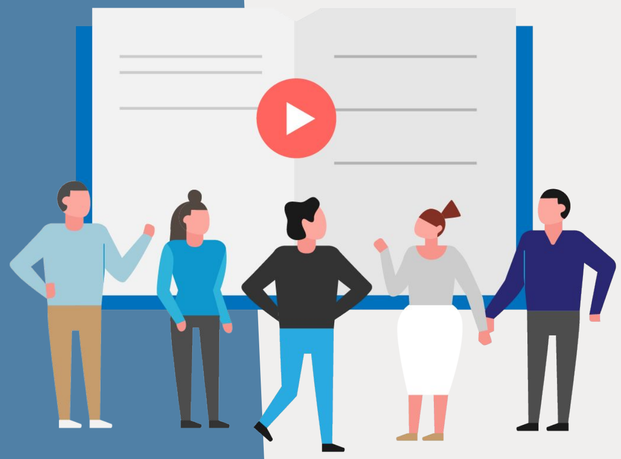
Many Side

One-to-Many Relationship



Student	
id	integer
first_name	text
last_name	text
age	integer
class_id	integer

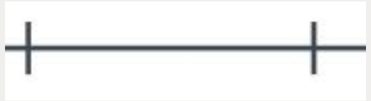
Class	
id	text
classroom	text
class_size	integer

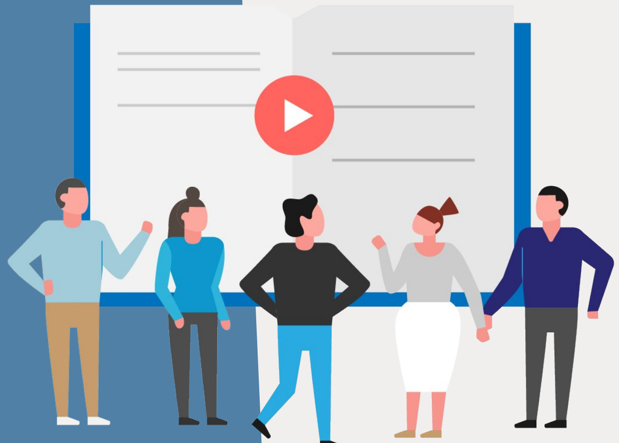


One-to-One Relationship

</talentlabs>

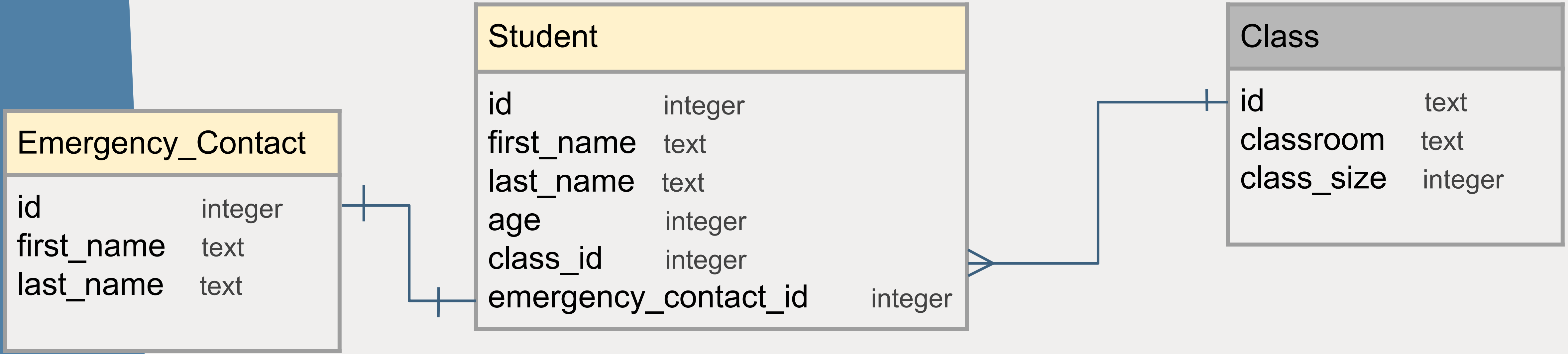
One-to-One Relationship

One Side  One Side



</talentlabs>

One-to-One Relationship



Emergency_Contact	
id	integer
first_name	text
last_name	text

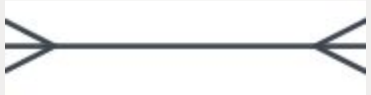
Student	
id	integer
first_name	text
last_name	text
age	integer
class_id	integer
emergency_contact_id	integer

Class	
id	text
classroom	text
class_size	integer

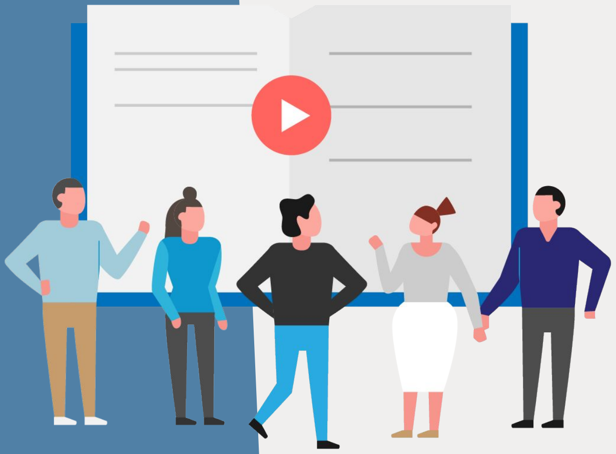
Many-to-Many Relationship

Many-to-Many Relationship

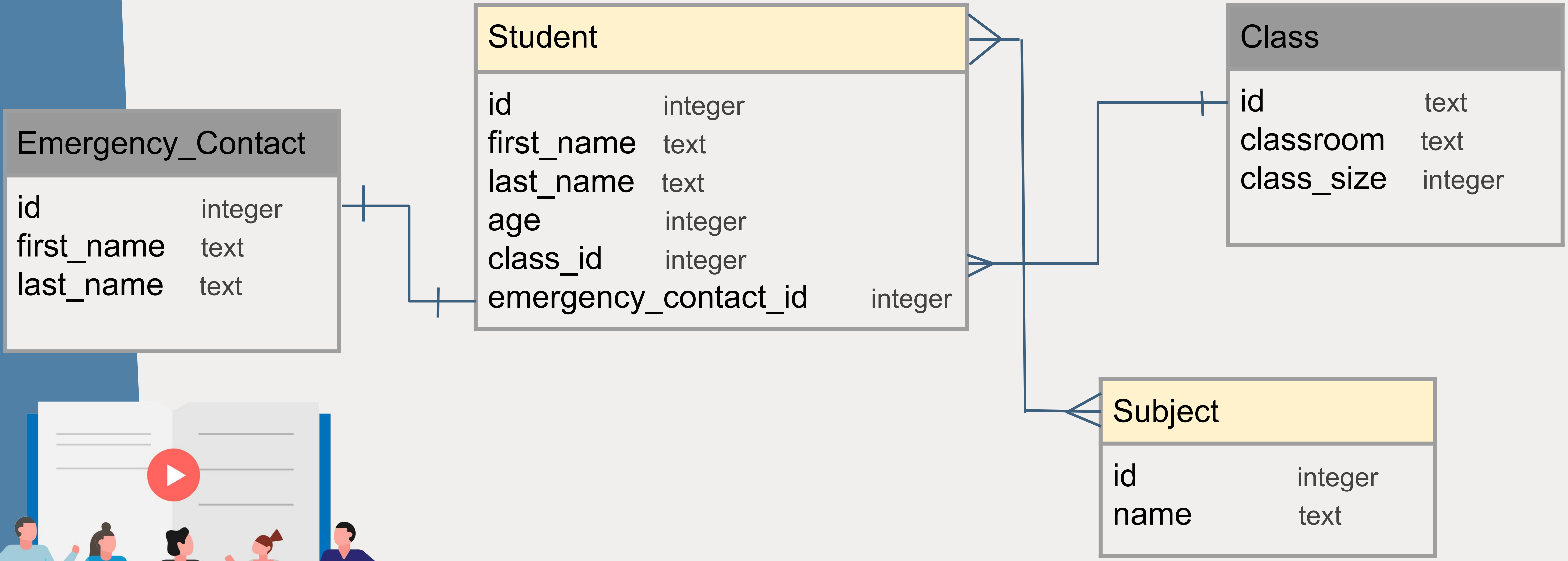
Many Side



Many Side



Many-to-Many Relationship (Simple Way)

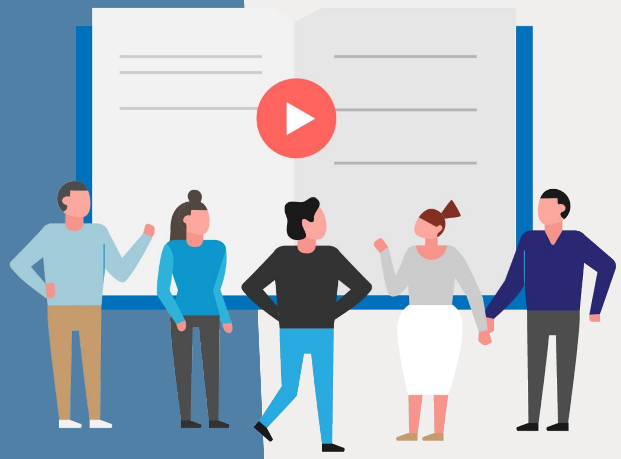


Student	
id	integer
first_name	text
last_name	text
age	integer
class_id	integer
emergency_contact_id	integer

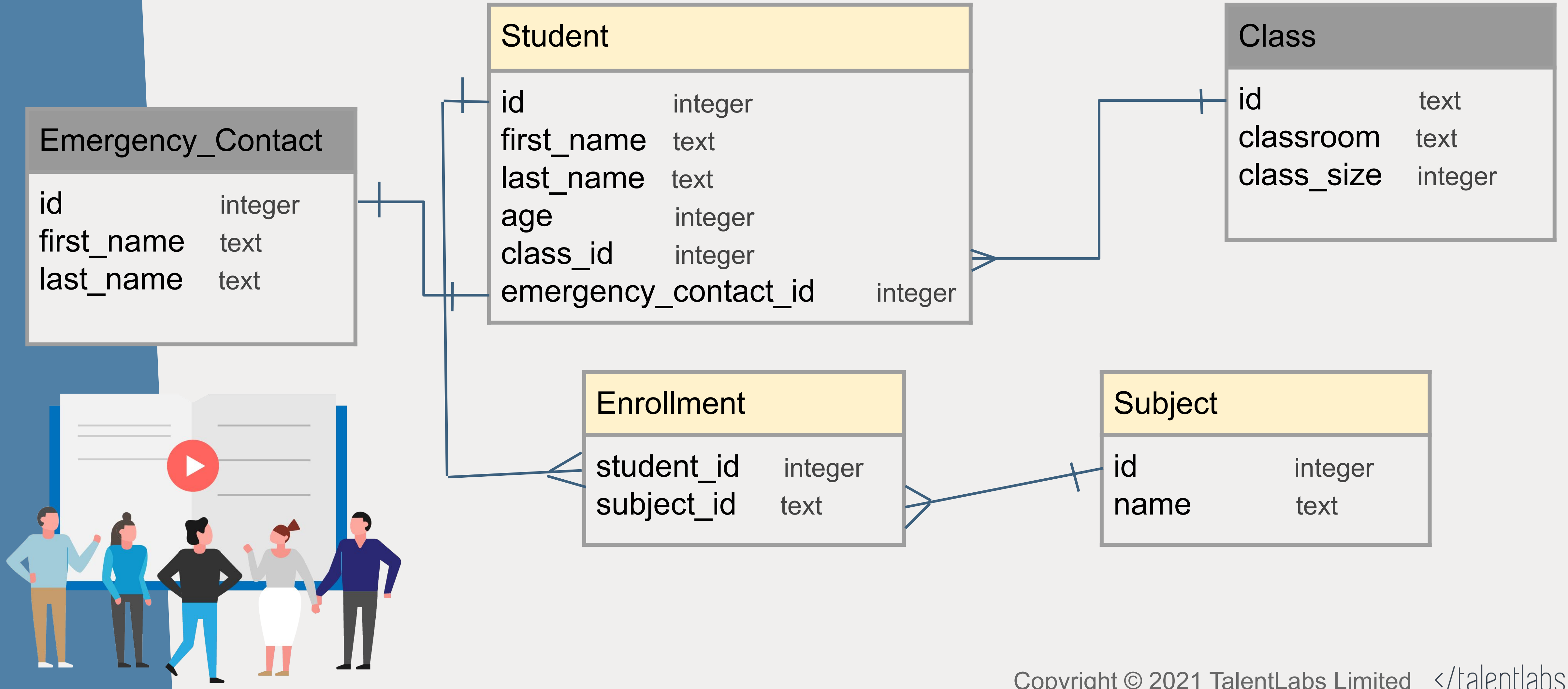
Emergency_Contact	
id	integer
first_name	text
last_name	text

Class	
id	text
classroom	text
class_size	integer

Subject	
id	integer
name	text



Many-to-Many Relationship (Alternative Way)



Student	
id	integer
first_name	text
last_name	text
age	integer
class_id	integer
emergency_contact_id	integer

Class	
id	text
classroom	text
class_size	integer

Emergency_Contact	
id	integer
first_name	text
last_name	text

Enrollment	
student_id	integer
subject_id	text

Subject	
id	integer
name	text

</talentlabs>