neueda

Core Database Concepts

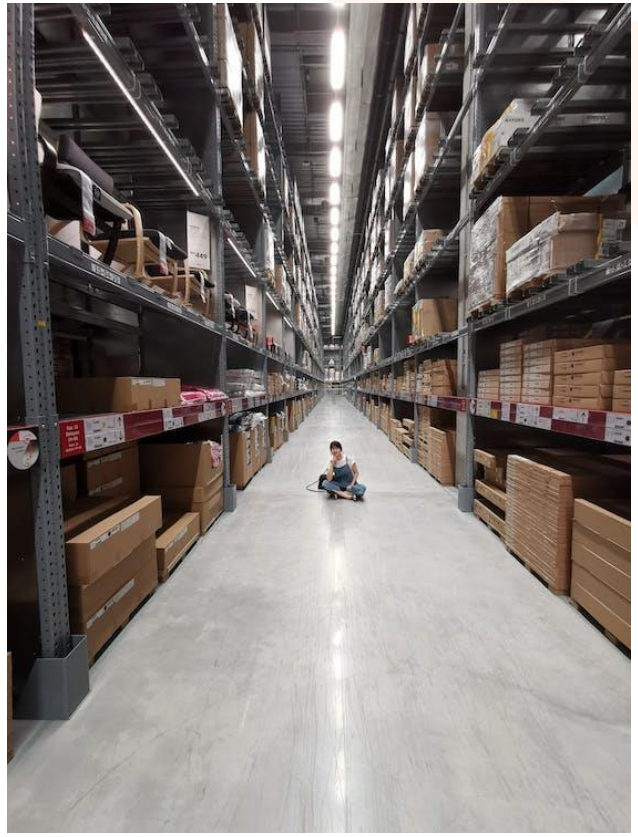
Objectives

- Core Database Terms
- Core MySQL Workbench Skills

Storing Data

- In the physical world, large amounts of 'stuff' is stored in a warehouse
  - These warehouses are well organised and arranged so that items can be easily retrieved and stored as required

- Where do companies store large amounts of 'digital stuff' or 'data'?
  - Data can be stored in a **relational database** which is also well-organised and allows for easy retrieval and storage



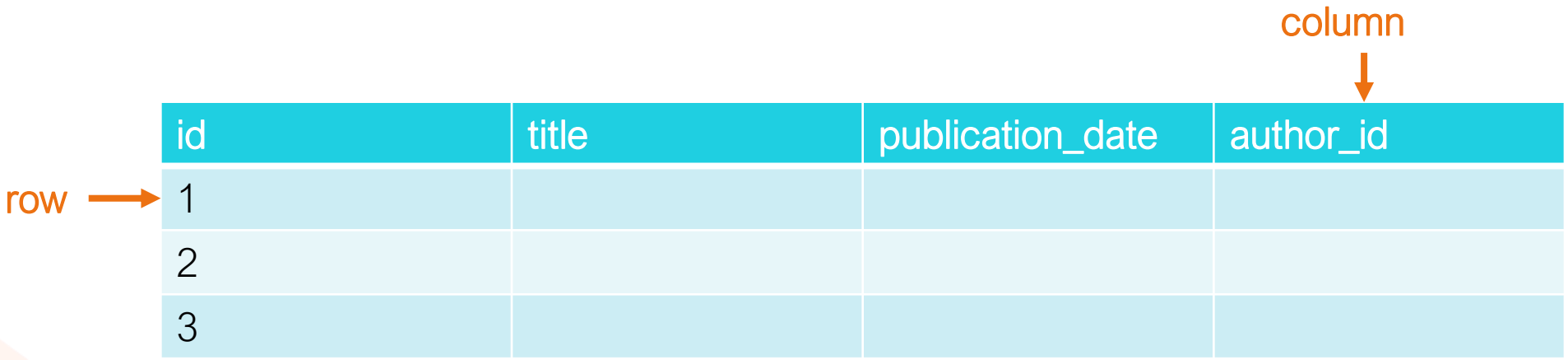
Core Database Terms

- When working with databases, some key terms are used
  - Table
  - Row
  - Column
  - Primary Key
  - CRUD
  - RDBMS
  - SQL

- You will be introduced to these key terms now

Table, Rows and Columns

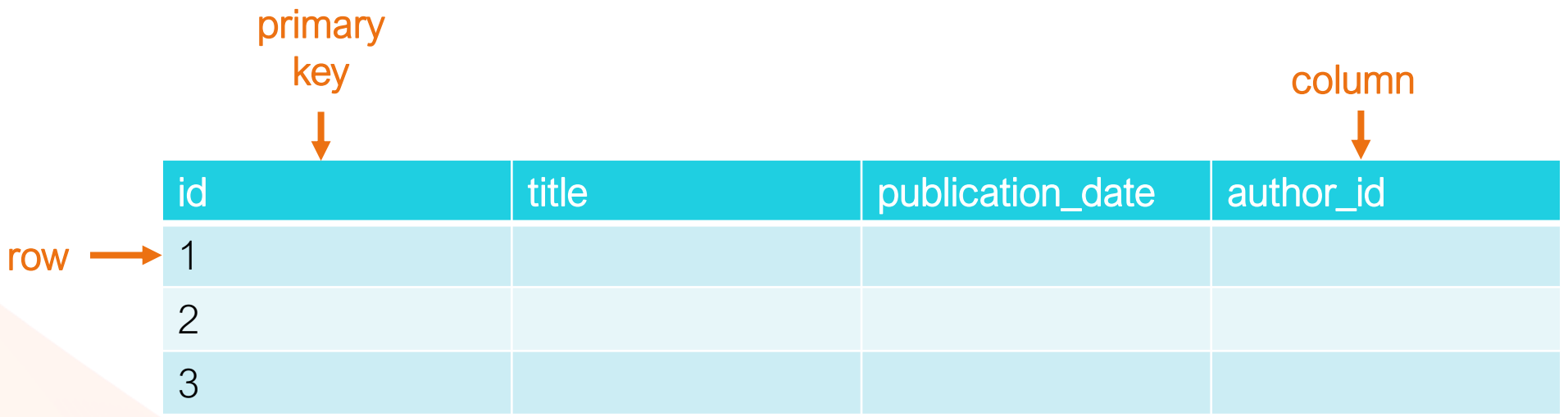
- In a relational database, all data is stored in **tables**
- For example, a table of books might look something like this



- Each item in the table is called a **row**
- Each row consists of a set of **columns**

Row Identifiers - Primary Keys

- Each row in a table has at least one column acting as the identifier or id
- This column is called the **primary key**



Interacting with the Database

- With any database, you almost always want to
  - Create
  - Read
  - Update
  - Delete
- This is sometimes referred to as **CRUD**

- Interactions are done using a syntax called **Structured Query Language (SQL)**
  - You will be introduced to the SQL language through the rest of the training
  - But where to use the SQL? You'll find out on the next slide

Database Management System (DBMS)

- A **Database Management System (DBMS)**
  - Is like the manager of our information warehouse
- There are two types of DBMS
    - Relational DBMS uses SQL, tables, rows, etc.
    - NoSQL systems which are out of the scope of this training

- A **DBMS** has
  - A backend storage system, as well as
  - A front-end user interface

- The **SQL** you learn works for each **RDBMS**
  - The SQL syntax may differ a bit between the systems
  - Some examples of the most popular systems are on the next slide

Examples of RDBMS Systems

- Commonly used RDBMS systems include
  - Oracle
  - SQL Server
  - Postgres
  - Sybase
  - MySQL
- Throughout the course, we will use
  - MySQL as the storage system
  - MySQL Workbench as the user interface
- The following few slides are about how to use MySQL workbench

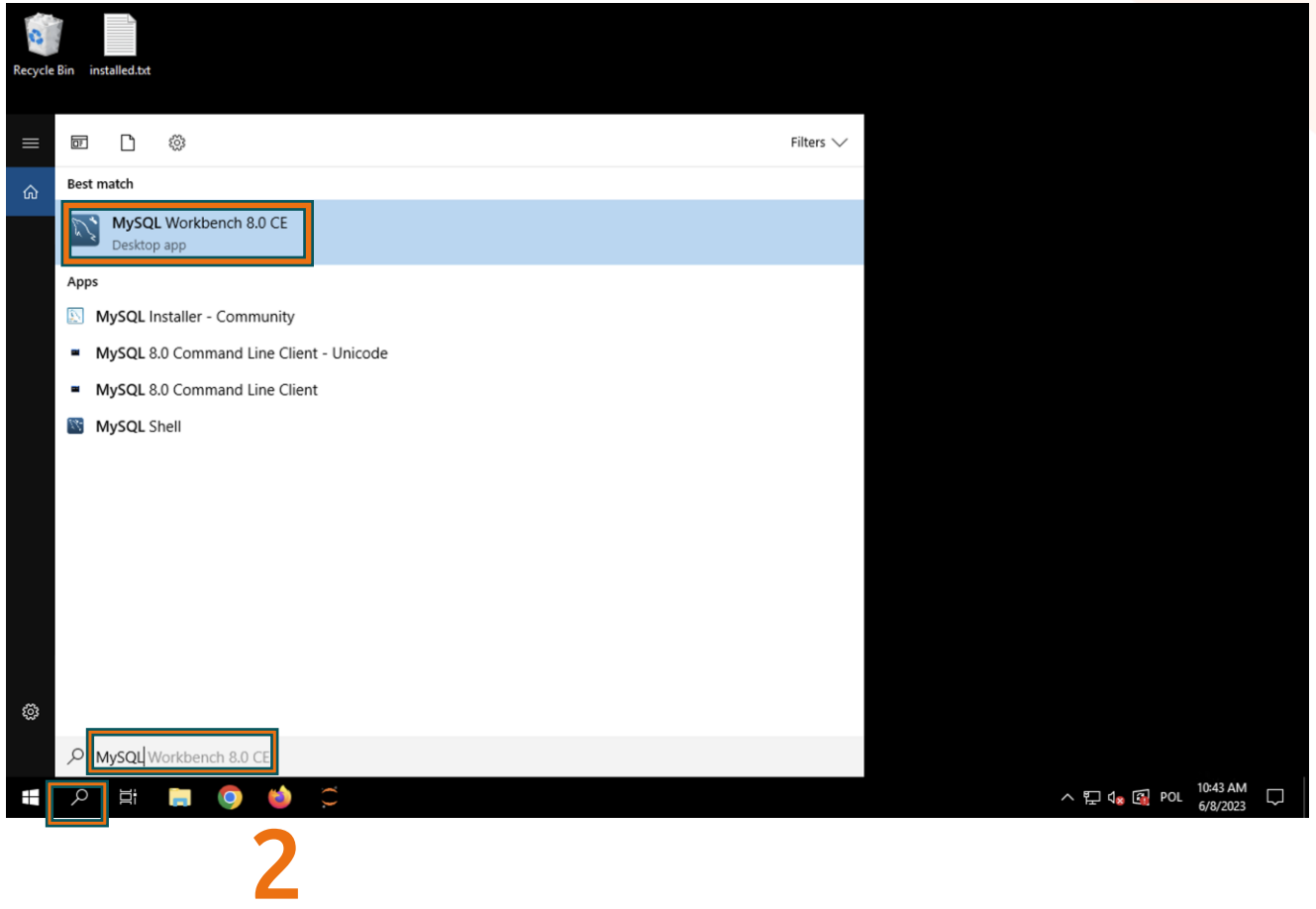
MySQL Workbench – Core Skills

- In the following slides, you will learn how to

1. Run the program
2. Login
3. Upload SQL file
4. Recognise The Key Parts of MySQL Workbench
5. Run SQL
6. Switch a database schema

1. MySQL Workbench – Run the program

1. In the Windows Operating System, Use the search box and
2. Type ‘MySQL’
3. Click MySQL Workbench

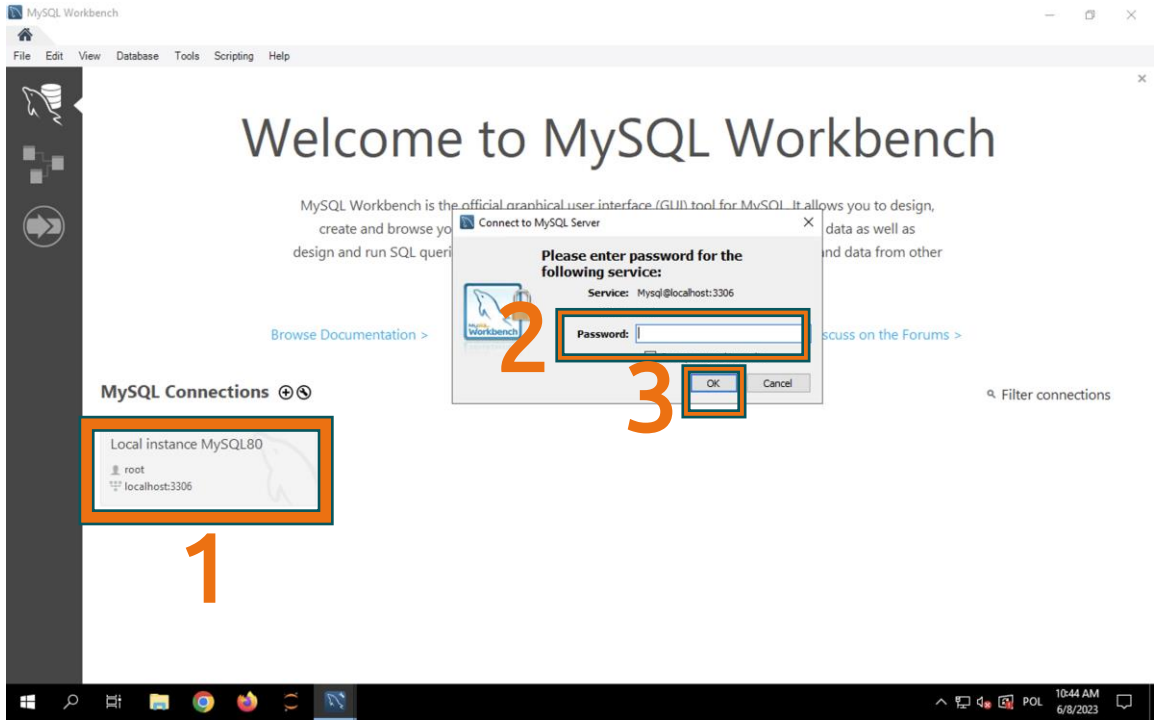


It will be different on alternative Operating Systems

neueda

2. MySQL Workbench – Login

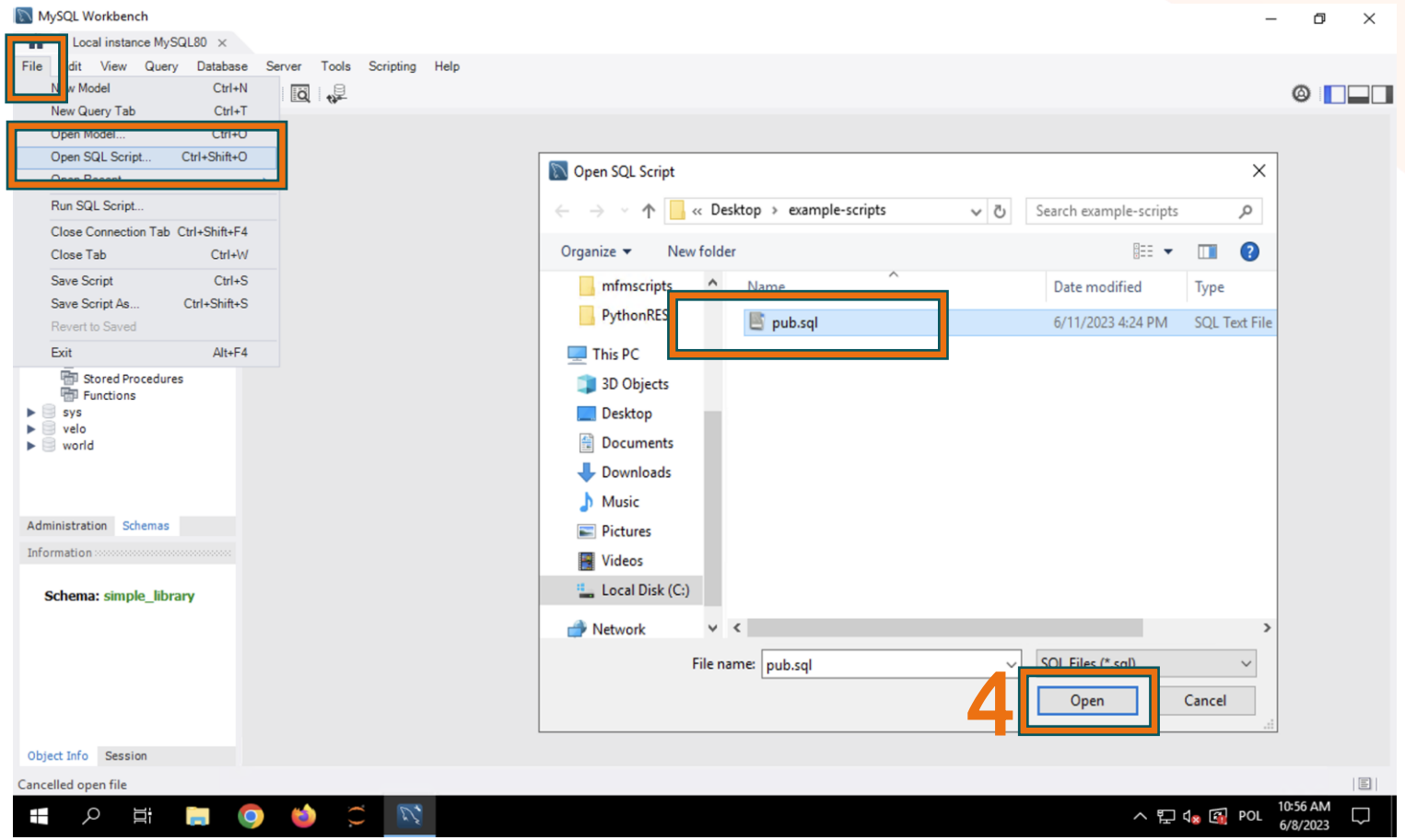
1. Click **local instance**
2. Type the password in the pop-up window
   - Ask your instructor for the password
3. Click **OK**



neueda

2. MySQL Workbench – Upload SQL file

1. Click **File**
2. Click **Open SQL Script**
3. Choose the file
4. Click **Open**



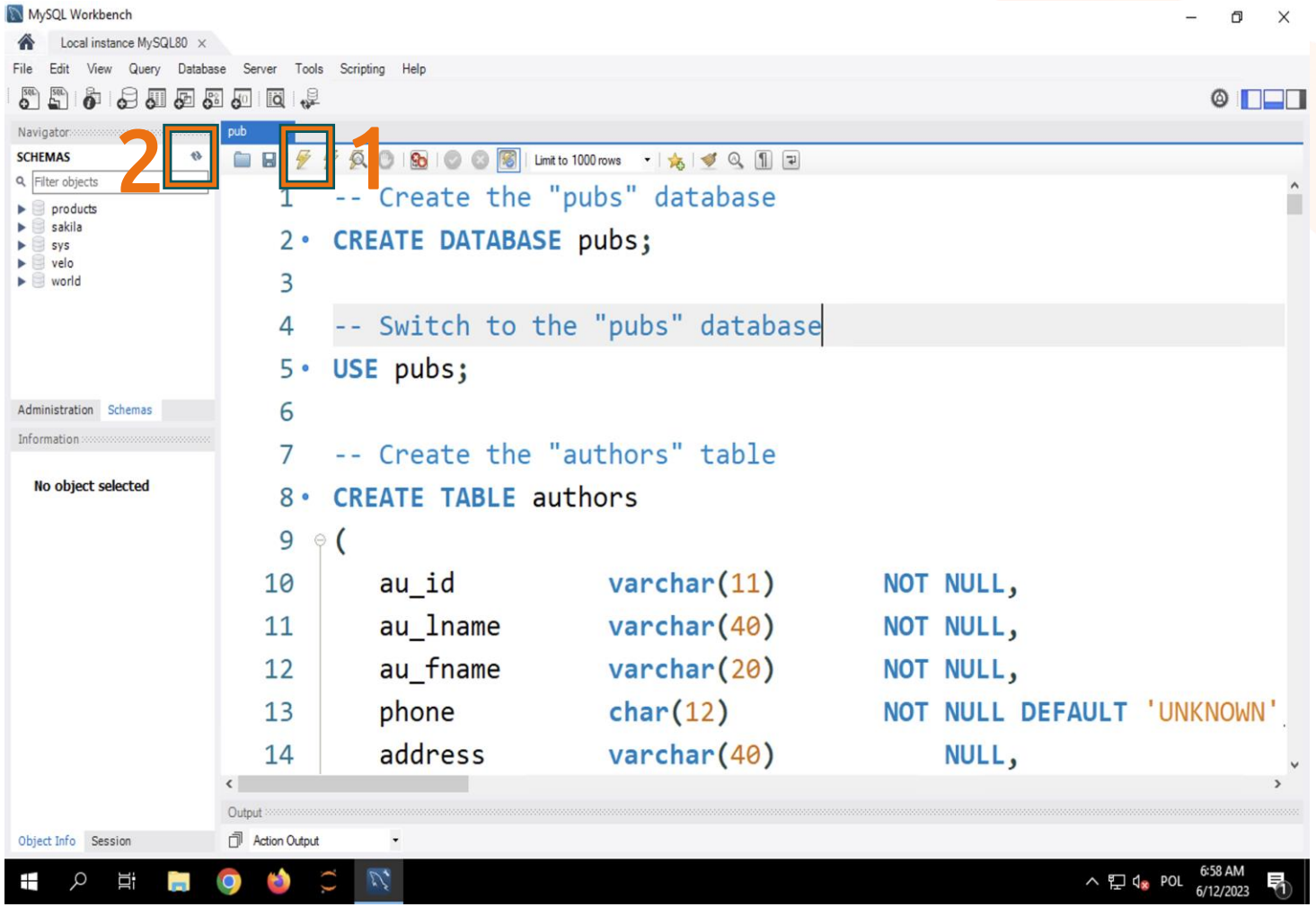
neueda

4. MySQL Workbench – Run SQL

- The SQL from the pub.sql file will appear in the editor pane

1. To run the SQL from the file, click on the lightning bolt as shown
2. In the SCHEMAS section, click on the refresh icon to see the pubs schema

- It's time to get familiar with MySQL Workbench. Afterwards, you'll be introduced to SQL



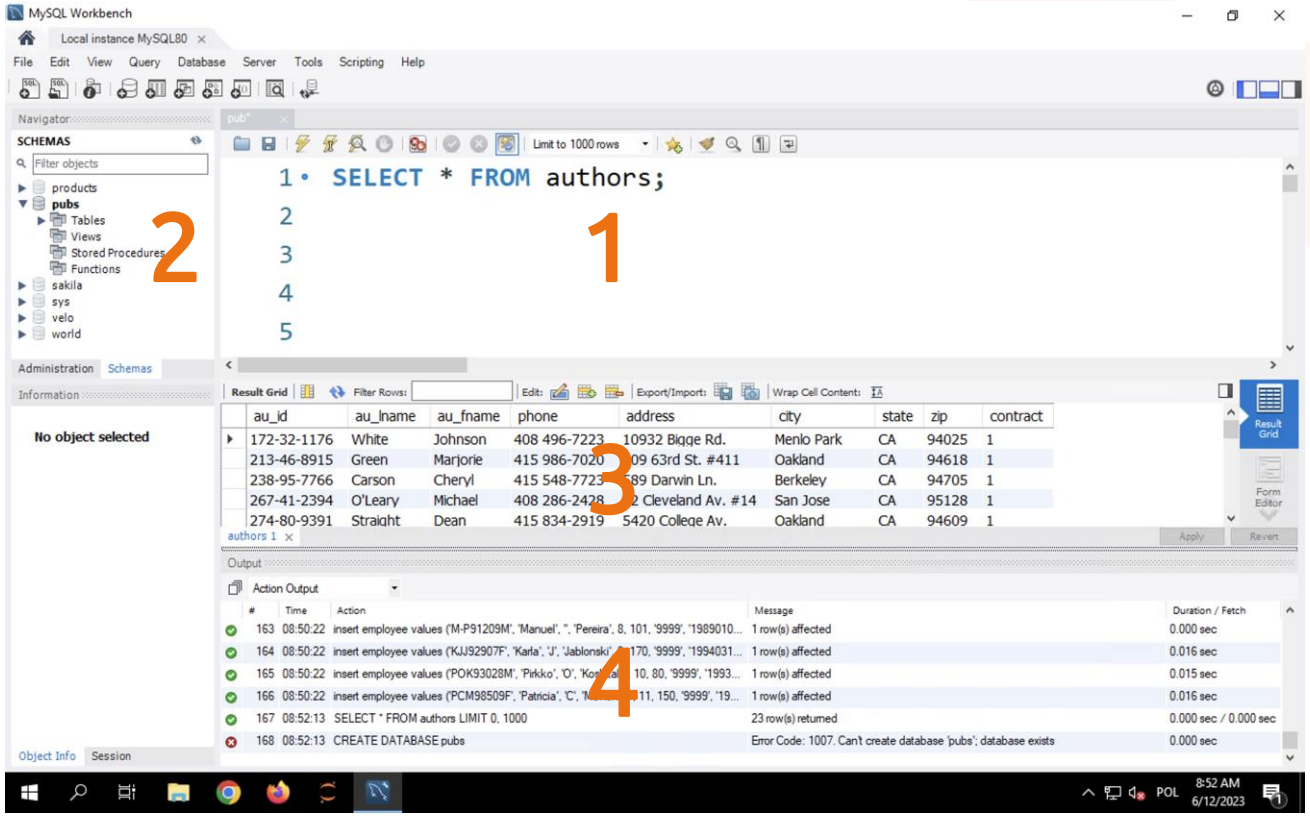
neueda

4. MySQL Workbench: First Time

- MySQL Workbench is a unified visual tool for database architects, developers, and Administrators
- It provides a graphical interface for managing and manipulating MySQL databases
- In the beginning, you may feel like it’s your first time in a cockpit of a plane
- Although MySQL Workbench has many features, most of the time, you’ll focus on 4 Key Parts, which are described in the following slide

4. MySQL Workbench: the Key Parts

1. **SQL Editor**: This is where you write and execute your SQL queries
2. **Schema Inspector**: This allows you to view and modify all the objects in your schema
   - (tables, views, stored procedures, etc.)
3. **Query Results Panel**: Your query results
4. **Output Panel**: This section displays executed queries' results,
   - including error messages

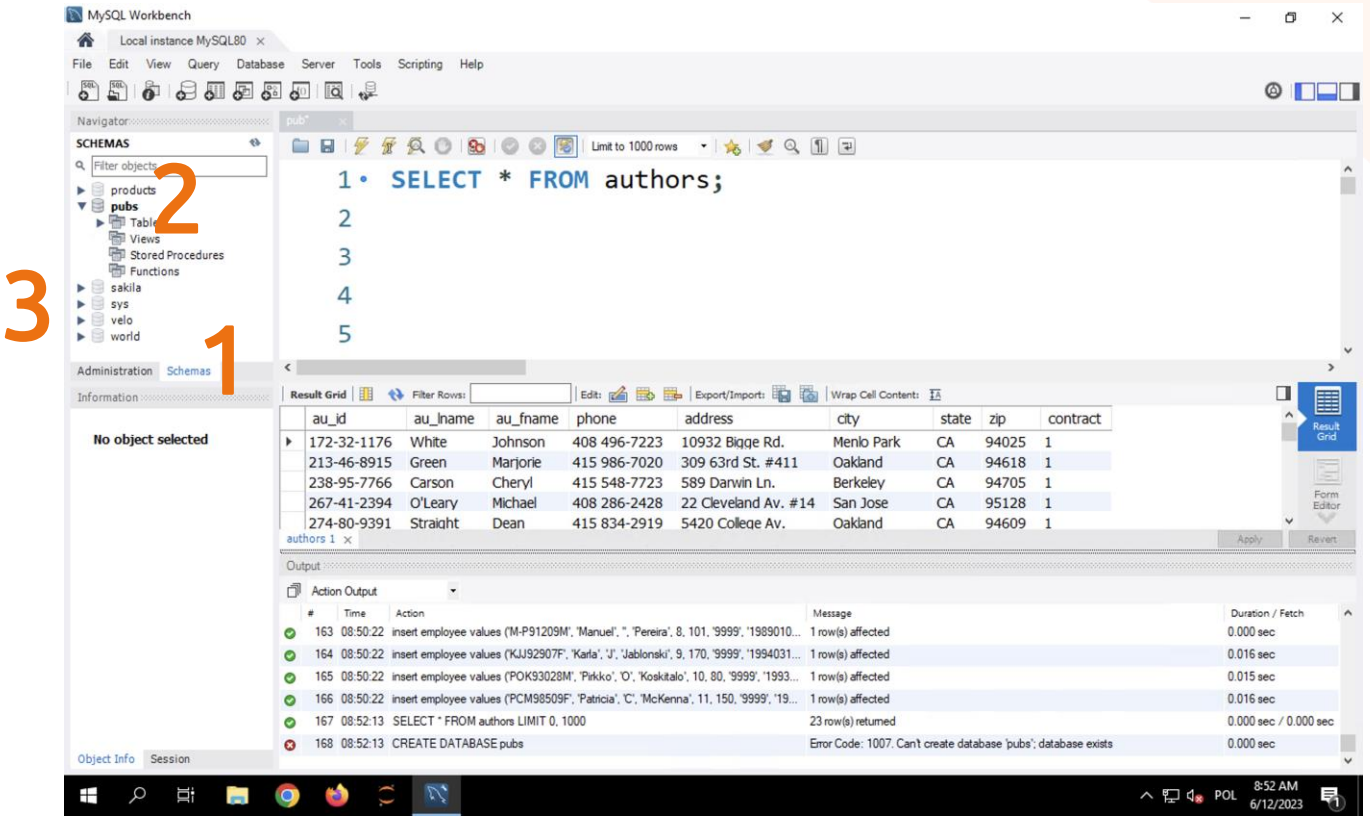


neueda

3. MySQL Workbench – Switch Schema

- Sometimes you need to change the current schema

1. In the **Schema Inspector**, click the **Schemas** tab
2. The current schema is highlighted in bold
3. To switch the active schema, double-click the schema you wish to use, e.g. world



neueda

What’s Next

- By now, you should be familiar with the following concepts & skills
  - Table, Row, Column
  - Primary Key
  - CRUD
  - RDBMS
  - MySQL Workbench
    - Recognise the key sections
    - Upload and run SQL
    - Switch schemas
- You are ready for your first SQL!

Summary

- Core Database Terms
- Core MySQL Workbench Skills

neueda

CRUD

READ (SELECT)

Objectives

- Reading the data with SELECT
- Controlling the Result Set
- Aliases
- Basic Arithmetic Operations
- SQL Syntax

Reading Data - using Select

- To read data from tables, you can use a **SELECT** SQL statement

```
SELECT * FROM authors;
```

- **SELECT** means to select some rows
- ***** means I want all the columns
- **FROM** specifies the table you want to retrieve data from
- **authors** is the name of the table

Result Set

- The result of a SQL **SELECT** statement is referred to either as a
  - **Result Set**
  - Set of rows

```
+-------------+-----------+----------+--------------+----------------------+-------------+------+-------+--------+
|    au_id    | au_lname  | au_fname |    phone     |       address        |     city    | state| zip   | status|
+-------------+-----------+----------+--------------+----------------------+-------------+------+-------+--------+
| 409-56-7008 | Bennet    | Abraham  | 415 658-9932 | 6223 Bateman St.     | Berkeley    | CA   | 94705 | 1      |
| 213-46-8915 | Green     | Marjorie | 415 986-7020 | 309 63rd St. #411    | Oakland     | CA   | 94618 | 1      |
| 238-95-7766 | Carson    | Cheryl   | 415 548-7723 | 589 Darwin Ln.       | Berkeley    | CA   | 94705 | 1      |
| 998-72-3567 | Ringer    | Albert   | 801 826-0752 | 67 Seventh Av.       | Salt Lake   | UT   | 84152 | 1      |
| 899-46-2035 | Ringer    | Anne     | 801 826-0752 | 67 Seventh Av.       | Salt Lake   | UT   | 84152 | 1      |
+-------------+-----------+----------+--------------+----------------------+-------------+------+-------+--------+

Note: The table above only shows the first five rows of data. There are more rows in the result set.
```

Manipulating the Result Set

- Using **SELECT *** is retrieving all the data from the table
  - Like retrieving an entire haystack when sometimes you just want a needle

- Often we just need specific information
  - Saving time and improving the accuracy of our results
- Let's look at some examples of how to do it

Controlling which Columns are Returned

- In the select statement, you can specify the **column names** you want in the result set

```
SELECT au_lname, au_fname FROM authors;
```

```
+-----------+----------+
| au_lname  | au_fname |
+-----------+----------+
| Bennet    | Abraham  |
| Green     | Marjorie |
| Carson    | Cheryl   |
| Ringer    | Albert   |
| Ringer    | Anne     |
+-----------+----------+

Note: The table above only shows the first five rows of
data. There are more rows in the result set.
```

Control Which Rows are Returned Using LIMIT

- **SELECT *** can return thousands or even millions of rows
- We can use **LIMIT** to, e.g.
  - Get the first 5 rows only
  - Skip the first 10 rows

```
-- Select all columns from the authors table and
--       limit the result set to 5 rows

SELECT * FROM authors LIMIT 5;


-- Select all columns from the authors table and
--      limit the result set to 5 rows (LIMIT)
--      skip the first 10 rows (OFFSET)

SELECT * FROM authors LIMIT 5 OFFSET 10;
```

Control Which Rows are Returned Using DISTINCT

- The **DISTINCT** keyword returns unique (distinct) values in the output
- It eliminates all duplicate records and fetches only the unique ones

```
-- SELECT DISTINCT column_name FROM table_name;

SELECT DISTINCT state FROM authors;
```

Control which Rows are Returned using WHERE

- With **WHERE** comes many possibilities for filtering data, among them:
  - =, <>, >, <, >=, <=
  - LIKE, IN, BETWEEN,
  - IS NOT NULL, IS NULL
  - AND, OR, NOT
- Unless you are doing an exam, you don’t need to memorise all of it
  - It will come with practice, and you can always check the documentation
- Let’s briefly look at some examples

SQL Comparison Operators

- Equal to: **=**
- Not equal to: **<>** or **!=**
- Greater than: **>**
- Less than: **<**
- Greater than or equal to: **>=**
- Less than or equal to: **<=**

```
-- Selects authors living in CA
SELECT * FROM authors WHERE state = 'CA';

-- Selects authors whose names are not 'Ringer'
SELECT * FROM authors WHERE au_lname <> 'Ringer';

-- Selects authors whose zip code is less than '95000'
SELECT * FROM authors WHERE zip < '95000';

-- Selects authors whose zip code is
-- greater than or equal to '94000'
SELECT * FROM authors WHERE zip >= '94000';
```

SQL Logical Operators

- **AND**
- **OR**

```
-- Authors called Heather
SELECT * FROM authors WHERE au_fname = 'Heather’;

-- Authors in TN called Heather
SELECT * FROM authors WHERE state = ‘TN' AND au_fname = 'Heather’;

-- Authors either in TN, or called Heather
SELECT * FROM authors WHERE state = 'TN’ OR au_fname = 'Heather' ;
```

Pattern Matching Operators

- **LIKE**: Performs a pattern match using wildcards
  - **%** (matches any sequence of characters) and
  - **_** (matches any single character)
- **IN**: Checks if a value matches any value in a list
- **BETWEEN**: Checks if a value is within a specified range

```
-- Names contain 'a' as the second letter and end with 'son'
SELECT * FROM authors WHERE author_name LIKE '_a%son';

-- Name is one of a list of names
SELECT * FROM authors WHERE au_fname IN ('Heather', 'Burt', 'Anne');

-- Titles priced between 2 and 10 dollars
SELECT * FROM titles WHERE price BETWEEN 2 AND 10;
```

NULL Comparison Operators

- **IS NULL**: Checks if a value is null
- **IS NOT NULL**: Checks if a value is not null

```
-- Retrieve authors who have a missing value for the degree column (NULL)
SELECT * FROM authors
WHERE degree IS NULL;

-- Retrieve authors who have a non-null value for the degree column
SELECT * FROM authors
WHERE degree IS NOT NULL;
```

Chain multiple WHERE operators

- You can **chain** multiple **WHERE operators** in SQL to create more complex conditions for filtering data.
- Here is an example on an **imaginary** students table (not in pubs database).

```
-- Retrieve students from the USA or Canada, with a degree of PhD or Masters,
-- whose names start with J or S, were born between 1980 and 2000,
-- and have a non-null city value.

SELECT * FROM students
WHERE (country = 'USA' OR country = 'Canada') -- Filter by country
  AND (degree = 'PhD' OR degree = 'Masters') -- Filter by degree
  AND (author_name LIKE 'J%' OR author_name LIKE 'S%') -- Filter by name pattern
  AND birth_year BETWEEN 1980 AND 2000 -- Filter by birth year range
  AND city IS NOT NULL; -- Filter by non-null city
```

Order the Result Set

- To ensure a specific ordering for your query results, you should
  - use the **ORDER BY** clause and
  - specify the column(s) you want to order by
    - **ASC**
    - **DESC**

```
SELECT * FROM authors ORDER BY au_id DESC;
```

SQL Aliases

- **Aliases** are particularly useful when
  - Handling complex queries
  - Performing operations on multiple tables, or
  - Renaming columns for the purpose of the output

```
SELECT author_id AS ID, au_fname AS FirstName, au_lname AS LastName
FROM authors;


+----+-----------+----------+
| ID | FirstName | LastName |
+----+-----------+----------+
|  1 | John      | Doe      |
|  2 | Jane      | Doe      |
+----+-----------+----------+
```

Basic Arithmetic Operations

- In MySQL, you can use, e.g. *** operator** to perform multiplication
- The same with the other operators: **+**, **-**, **/** and **%**
- We will cover the built-in functions later on

```
-- Calculating the discounted price after applying a 10% discount
SELECT title, price, price * 0.9 as 'discounted_price' FROM titles;

+----------------------------------+-------+------------------+
| title                            | price | discounted_price |
+----------------------------------+-------+------------------+
| Secrets of Silicon Valley        | 20.00 | 18.00            |
| The Busy Executive's Database... | 19.99 | 17.99            |
| Emotional Security: A New Algo.. | 7.99  | 7.19             |
+----------------------------------+-------+------------------+
```

SQL Syntax

- As you can see in the examples, **SQL** is pretty straightforward
  - You need a bit of practice to get comfortable with its syntax

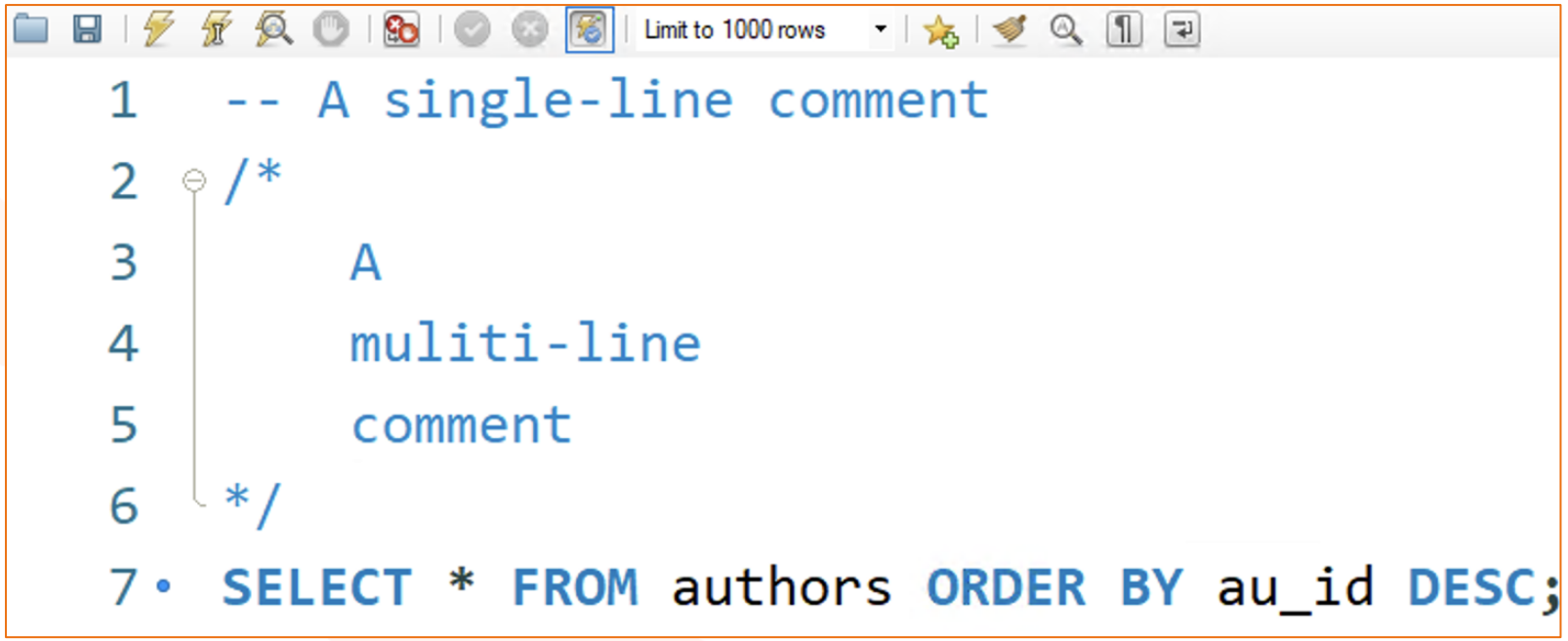
- In the examples, you might have noticed a few things not yet explained
  - Using the **semicolon**
  - Using the **code comments**
  - Some parts of the SQL are in all capital letters

- It’s time to get familiar with those bits

SQL Syntax: using the Code Comments

- Comments are used to explain SQL code
  - The compiler ignores them

- They can be single-line or multi-line



neueda

SQL Syntax: using the Semicolon

- In **SQL**, **semicolons** are used to **indicate** the **end of a statement**
- It's **essential** to use semicolons **when** you have **multiple statements** in a single **SQL** script
- Can you spot what is wrong with the below queries?

```
1  SELECT *
2  FROM authors
3  ORDER BY DESC;
```

```
1  SELECT * FROM authors;
2  SELECT * FROM stores
3  SELECT * FROM sales;
```

SQL Syntax: Case-Sensitivity

- **SQL keywords** (**SELECT**, **FROM**, **WHERE**, etc.) are not case-sensitive
  - In practice, you should follow your project's **SQL** convention
    - e.g. pick **SELECT** or **select** and use it all the time

- **Table and Column Names**
  - Different DBMSs work differently
    - MySQL on Windows is case-insensitive by default

- **Column Values**
  - **MySQL** have **case-insensitive** behaviours **by default**
    - **WHERE column = 'abc'** returns **TRUE** for e.g., **'abc', 'ABC', 'aBc'**

Summary

- Reading the data with SELECT
- Controlling the Result Set
- Aliases
- Basic Arithmetic Operations
- SQL Syntax

neueda

CRUD

CREATE, UPDATE, DELETE

Objectives

- Create: INSERT
- Update: UPDATE
- Delete: DELETE

Creating Data - using INSERT

- The **INSERT** statement adds new rows or records to a table

```
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3);
```

- **INSERT INTO** insert data into a table
- **table_name** is the name of the table where the data will be inserted
- **column1, column2, column3** are the **names of the columns** where the values will be inserted
- **VALUES** the values to be inserted
- **value1**, **value2, value3** provides **the actual values** that will be inserted into the corresponding columns

INSERT examples 1 of 4

- **INSERT** a new author with **all column values** specified

```
INSERT INTO authors (au_id, au_lname, au_fname, phone, address, city, state, zip, contract)
VALUES ('123-45-6789', 'Smith', 'John', '555-123-4567', '123 Main St.', 'New York', 'NY', '10001', 1);
```

- The above example
  - Inserts a new author
  - With the specified values for each column
  - All columns are provided with corresponding values

INSERT examples 2 of 4

- **INSERT** a new author **with some column values** specified

```
INSERT INTO authors (au_id, au_lname, au_fname)
VALUES ('987-65-4321', 'Doe', 'Jane');
```

- In the above example
  - only the **au_id**, **au_lname**, and **au_fname** columns are specified with values
  - The other columns will be
    - Assigned their default values or
    - Remain NULL if no default value is defined

INSERT examples 3 of 4

- **INSERT multiple rows** in a single statement

```
INSERT INTO authors (au_id, au_lname, au_fname)
VALUES
    ('111-11-1111', 'Williams', 'Michael'),
    ('222-22-2222', 'Johnson', 'Sarah'),
    ('333-33-3333', 'Brown', 'David');
```

- The above example demonstrates
  - Inserting multiple authors in a single **INSERT** statement
  - Each set of values within the parentheses
    - Represents a new row to be inserted

INSERT examples 4 of 4

- **INSERT** data from a **SELECT** statement

```
INSERT INTO authors (au_id, au_lname, au_fname)
SELECT emp_id, lname, fname
FROM employees
WHERE job_id = 1;
```

- In this example, the **INSERT** statement is used with a **SELECT** statement
- It inserts data into the **authors** table by
  - Selecting specific columns from the **employees** table
    - Based on a condition

Modify Existing Data - using UPDATE

- To modify existing data, you can use an **UPDATE** SQL statement

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

- **UPDATE** is the keyword that indicates we want to update records in a table
- **table_name** is the name of the table where the records will be updated
- **SET** is used to specify the columns to update and their new values
- **WHERE** is an optional clause that allows you to specify conditions to filter the rows that will be updated
- Some examples are provided on the following slides

UPDATE examples 1 of 3

- Updating a Single Column

```
-- Update the phone number of an author with a specific ID
UPDATE authors
SET phone = '555-123-4567'
WHERE au_id = '409-56-7008';
```

- Updating Multiple Columns

```
-- Update the phone and address of an author with a specific ID
UPDATE authors
SET phone = '555-987-6543', address = '123 Main St'
WHERE au_id = '409-56-7008';
```

UPDATE examples 2 of 3

- Updating Rows based on a Condition

```
-- Update the contract status of all authors with a specific city
UPDATE authors
SET contract = 0
WHERE city = 'Oakland';
```

- Updating Rows with Complex Conditions

```
-- Update the contract status of authors who have sold more than 5000 copies
-- or have a royalty percentage greater than 15
UPDATE authors
SET contract = 0
WHERE (ytd_sales > 5000 OR royaltyper > 15) AND contract = 1;
```

UPDATE examples 3 of 3

- Updating Rows using Subquery

```
-- Update the contract status of authors who have sold
-- more than the average number of copies
UPDATE authors
SET contract = 0
WHERE au_id IN (
   SELECT au_id
   FROM titleauthor
   GROUP BY au_id
   HAVING SUM(qty) > (
      SELECT AVG(ytd_sales)
      FROM titles
      WHERE type = 'popular_comp'
   )
);
```

- Don’t bother with the query details for now – it’s just an example of an update with a bit more complex subquery

Deleting Existing Data - using DELETE

- The **DELETE** command is used to remove one or more rows from a table

```
DELETE FROM table_name
WHERE condition;
```

- **DELETE FROM** specifies the table from which you want to delete rows
- **WHERE** is optional and allows you to filter the rows to be deleted
  - based on specific criteria

- Now that you know the syntax, it's time to see some examples

Deleting Existing Data - examples

- Deleting a Single Row

```
-- Delete an author
-- with a specific author ID
DELETE FROM authors
WHERE au_id = '409-56-7008';
```

- Deleting Multiple Rows

```
-- Delete authors who are
-- not under contract
DELETE FROM authors
WHERE contract = 0;
```

- Deleting Rows Based on a Subquery

```
-- Delete authors who
-- have not made any sales
DELETE FROM authors
WHERE au_id NOT IN (
   SELECT au_id
   FROM titleauthor
);
```

Deleting Existing Data – Warning

- The **DELETE** command can remove all rows from the table, resulting in data loss
  - **DON'T DO THIS UNLESS YOU WANT TO COMPLETELY EMPTY A TABLE!!**

```
DELETE FROM table_name;
```

- Always double-check the condition and ensure it accurately identifies the rows you want to delete

- It's recommended first to perform a **SELECT** query with the same condition to verify the rows that will be affected before executing the **DELETE** command

- Take regular database backups to mitigate the risk of accidental data deletion

Summary

- Create: INSERT
- Update: UPDATE
- Delete: DELETE

neueda

AGGREGATE FUNCTIONS

Objectives

- Aggregate Functions
- GROUP BY and HAVING

Aggregate Functions

- Aggregate functions in SQL
  - Allow you to perform calculations on sets of rows and
    - **Return a single value**
  - They are commonly used to
    - **Summarise data** and
    - **Provide insights** into the dataset

- In the following slides, you will see some examples of
  - **COUNT()**
  - **SUM()**
  - **AVG()**
  - **MIN()**
  - **MAX()**
  - **GROUP BY and HAVING**

COUNT()

- The **COUNT()** function **counts the number of rows**
  - That match a specific criterion
- In the example below, we count
  - The total number of authors in the "authors" table

```
-- Count the total number of authors
SELECT COUNT(*) AS total_authors
FROM authors;

Result Set Example:
+---------------+
| total_authors |
+---------------+
|      5        |
+---------------+
```

SUM()

- The **SUM()** function calculates **the sum of a numeric column**
- The example below calculates
  - The total sales amount from the "sales_data" table

```
-- Calculate the total sales amount

SELECT SUM(sales) AS total_sales
FROM sales_data;

Result Set Example:
+-------------+
| total_sales |
+-------------+
|   24500     |
+-------------+
```

AVG()

- The **AVG()** function calculates **the average value of a numeric column**
- In the example below, we calculate
  - The average price of books from the ”books" table

```
-- Calculate the average price of books

SELECT AVG(price) AS average_price
FROM books;

Result Set Example:
+-----------------+
| average_price   |
+-----------------+
|     25.75       |
+-----------------+
```

MIN()

- The **MIN()** function retrieves **the minimum value from a column**
- In the example below, we find
  - The minimum price of books from the ”books" table

```
-- Find the minimum price of books

SELECT MIN(price) AS min_price
FROM books;

Result Set Example:
+-----------+
| min_price |
+-----------+
|   15.99   |
+-----------+
```

MAX()

- The **MAX()** function retrieves the **maximum value from a column**
- The example below shows
  - The maximum salary of employees from the "authors" table

```
-- Find the maximum salary of authors

SELECT MAX(salary) AS max_salary
FROM authors;

+--------------+
| max_salary   |
+--------------+
|   75000      |
+--------------+
```

GROUP BY

- The **GROUP BY** clause **groups rows**
  - Based on a specific column or columns

```
-- 1. Retrieve the author ID (au_id) and the count of books associated with each author
-- 2.    The COUNT(*)  function counts the number of books for each author ID
-- 3.     Group the rows by au_id using the GROUP BY clause
-- Authors with no books will have a count of 0

SELECT au_id, COUNT(*) AS book_count
FROM titleauthor
GROUP BY au_id;

Result Set Example:
+------------+------------+
|   au_id    | book_count |
+------------+------------+
| 172-32-1176|     2      |
| 213-46-8915|     1      |
| 267-41-2394|     3      |
| 409-56-7008|     0      |
+------------+------------+
```

HAVING

- **HAVING** is similar to the **WHERE** clause but used explicitly with aggregate functions
  - Such as COUNT, SUM, AVG, etc.

1. **GROUP BY** creates distinct groups in the result set
2. **Aggregate Functions** are applied to these groups, calculating values for each group, e.g. **COUNT**
3. **HAVING** is then used to filter the groups based on conditions applied to the aggregated values

```
-- The same SQL as on the previous slide
--  but this time, with HAVING

SELECT au_id, COUNT(*) AS book_count
FROM titleauthor
GROUP BY au_id;
HAVING book_count >= 2;

Result Set Example:
+------------+------------+
|   au_id    | book_count |
+------------+------------+
| 172-32-1176|     2      |
| 267-41-2394|     3      |
+------------+------------+
```

HAVING continued

```
-- Selecting from the table:
SELECT au_id, au_fname, au_lname
FROM authors;
```

```
+-----------+---------+----------+
|  au_id    | au_fname | au_lname |
+-----------+---------+----------+
| 172-32-1176| Johnson  | White    |
| 213-46-8915| Marjorie | Green    |
| 267-41-2394| Michael  | O'Leary  |
| 409-56-7008| Abraham  | Bennet   |
| 648-92-1872| Reginald | Blotchet |
+-----------+---------+----------+
```

```
-- Grouping by a column
SELECT au_id, COUNT(*) AS book_count
FROM titleauthor
GROUP BY au_id;
```

```
+-----------+------------+
|  au_id    | book_count |
+-----------+------------+
| 172-32-1176| 3          |
| 213-46-8915| 2          |
| 267-41-2394| 1          |
| 409-56-7008| 0          |
| 648-92-1872| 0          |
+-----------+------------+
```

```
-- Filter with HAVING
SELECT au_id, COUNT(*) AS book_count
FROM titleauthor
GROUP BY au_id
HAVING book_count >= 2;
```

```
+-----------+------------+
|  au_id    | book_count |
+-----------+------------+
| 172-32-1176| 3          |
| 213-46-8915| 2          |
+-----------+------------+
```

Summary

- Aggregate Functions
- GROUP BY and HAVING

neueda

JOINs

Objectives

- Why joins are performed
- How rows are matched using a reference column
- Join types and their use cases:
  - INNER JOIN
  - LEFT JOIN
  - RIGHT JOIN
  - FULL JOIN (UNION)

Why Joins?

- Imagine you wanted to get an author name and then the title of a book
  - The author information is in the author table
  - The book information is in the book table
- How would you do that?
  - This is where joins come in

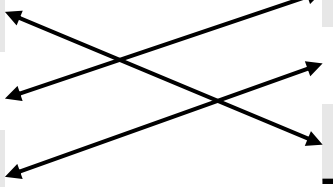
- You want information that is typically coming from more than one table

Read Data From Multiple Tables using JOIN

- **JOIN** allows you to retrieve data from multiple tables in a single query.
- Rows are matched across tables based on a reference column.

Table 1

words_1	ref_col
Let's	1
Joins	5
More	3



ref_col	words_2
5	Yay
3	About
1	Learn

Table 2

Example Join Result, Table 1 and Table 2

ref_col	words_1	words_2
1	Let's	Learn
3	More	About
5	Joins	Yay

Join Examples: The Database

- Example queries in this slide deck use the sports_joins database.
- It's a simplified example of an equipment inventory for a sports club.
- Here are the tables:

```
SELECT * FROM sports
```

```
-------------------------------------
sport_id sport            team_size
-------------------------------------
1        Football         11
2        Cricket          11
3        Ice Hockey       18
4        Tennis           1
5        Netball          7
-------------------------------------
```

```
SELECT * FROM equipment
```

```
-------------------------------------
item_id item_name         sport_id
-------------------------------------
1       Ball              1
2       Racket            4
3       Pommel Horse      7
4       Puck              3
5       Hockey stick      3
-------------------------------------
```

Query: Which Sports do we Offer?

- The sports table logs the sports that are advertised by the club, and that the club has permission to host.

```
SELECT * FROM sports



-------------------------------------
sport_id sport            team_size
-------------------------------------
1        Football         11
2        Cricket          11
3        Ice Hockey       18
4        Tennis           1
5        Netball          7
-------------------------------------
```

Query: What Equipment do we Have in Stock?

- The equipment table logs items in stock.
- Each item has a sport_id, which corresponds to the sport that the item is used for.

```
SELECT * FROM equipment



-------------------------------------
item_id item_name         sport_id
-------------------------------------
1       Ball              1
2       Racket            4
3       Pommel Horse      7
4       Puck              3
5       Hockey stick      3
-------------------------------------
```

Query: Which Sporting Events Could we Host?

- We can only host sports that the club **both**:
  - Officially offers / has permission to host (sports table)
  - Has equipment for (equipment table)

- We need to use a **join** because we need information from both tables.
- We will start with **theory** of joins before we look how to write them.

Joining ON a Column

- Joins are always performed **on** a column that allows rows to be matched across tables.
- In this example, the tables will be joined **ON** sport_id

sport_id	sport	team_size
1	Football	11
2	Cricket	11
3	Ice Hockey	18
4	Tennis	1
5	Netball	7

item_id	item_name	**sport_id**
1	Ball	1
2	Racket	4
3	Pommel Horse	7
4	Puck	3
5	Hockey stick	3

Joining ON a Column to Match Rows

- Joining ON sport_id allows rows to be matched across tables
- The sport of tennis can be matched with the item racket : they both have **sport_id = 4**

Example, showing just one row from a join between the tables sports and equipment **ON** sport_id:

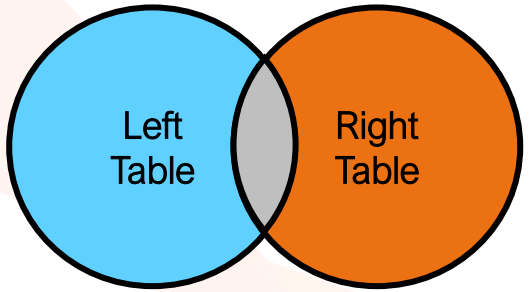
```
-- Columns from sports table          -- Columns from equipment table

------------------------------------  ------------------------------------
item_id item_name        sport_id     sport_id sport           team_size
------------------------------------  ------------------------------------
2   Racket               4 <------->  4        Tennis          1
------------------------------------  ------------------------------------
```

Join Types

- There are different types of **JOIN** operations, including:
  - **INNER JOIN**
  - **LEFT JOIN**
  - **RIGHT JOIN**
  - **FULL JOIN**

- The type of join decides what happens to any unmatched rows.

Full Join



- **FULL JOIN** returns
  - **All rows** from both tables, **including unmatched** rows
  - If there is **no match**, **NULL** values are included

```
Example FULL JOIN query result:


--columns from sports     --columns from equipment
-------------------------------------
sport_id sport            sport_id item_name
-------------------------------------
1        Football         1        Ball
2        Cricket          NULL     NULL
3        Ice Hockey       3        Hockey stick
3        Ice Hockey       3        Puck
4        Tennis           4        Racket
5        Netball          NULL     NULL
NULL     NULL             7        Pommel Horse
```


neueda

What are Unmatched Rows?

- Unmatched rows occur when either table has a value in the reference column (e.g. sport_id) that does not exist in the other table.

Unmatched row: there is no sport in sports with a **sport_id** of **7**

i.e. we have a pommel horse but don't officially offer gymnastics events.

```
Full join query result:


--columns from sports     --columns from equipment
-------------------------------------
sport_id sport            sport_id item_name
-------------------------------------
1        Football         1        Ball
2        Cricket          NULL     NULL
3        Ice Hockey       3        Hockey stick
3        Ice Hockey       3        Puck
4        Tennis           4        Racket
5        Netball          NULL     NULL
NULL     NULL             7        Pommel Horse
```

Unmatched row:

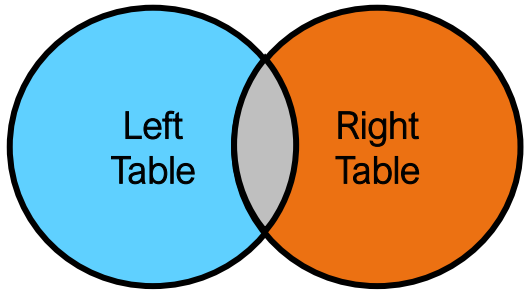
There is no row in equipment with a **sport_id** of **2**

i.e. we have no bat for our cricket matches.

Which Sporting Events Could we Host?-> Full Join?

The correct type of JOIN for the scenario depends whether we want to retrieve **unmatched** rows.

Scenario: given the sports we offer, and the equipment that's available, which sporting events could we host? Do we need the mismatched rows to answer this question?




```
Full join query result:


--columns from sports     --columns from equipment
-------------------------------------
sport_id sport            sport_id item_name
-------------------------------------
1        Football         1        Ball
2        Cricket          NULL     NULL
3        Ice Hockey       3        Hockey stick
3        Ice Hockey       3        Puck
4        Tennis           4        Racket
5        Netball          NULL     NULL
NULL     NULL             7        Pommel Horse
```

Are the Unmatched Rows Useful for the Scenario?

Scenario: given the sports we offer, and the equipment that's available, which sporting events could we host?

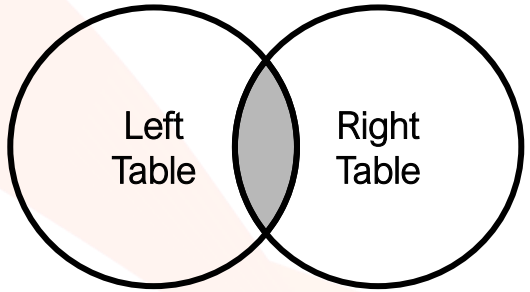
- Can we host sports for which we do not have equipment? **NO**
  - Can't play cricket with no bat
  - Can't play netball without the right type of ball
- Can we use equipment for sports that we do not officially offer? **NO**
  - Can't use the pommel horse if we don't have permission to host a gymnastics event

For this scenario, we probably don't want to retrieve the unmatched rows.

Example full join query result:

--columns from sports		--columns from equipment	
sport_id	sport	sport_id	item_name
1	Football	1	Ball
2	Cricket	NULL	NULL
3	Ice Hockey	3	Hockey stick
3	Ice Hockey	3	Puck
4	Tennis	4	Racket
5	Netball	NULL	NULL
NULL	NULL	7	Pommel Horse

Inner Join



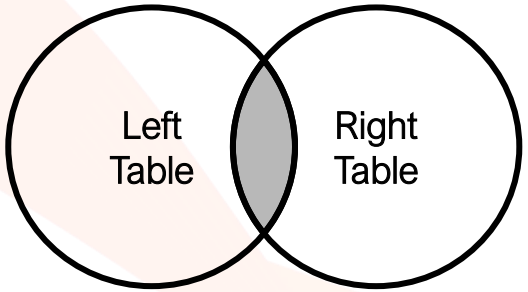
- **INNER JOIN** returns
  - **Only** rows that **match** based on the chosen column (e.g. sport_id) **in both tables**

```
-- Example INNER JOIN query result

-------------------------------------
sport_id sport           item_name
-------------------------------------
1        Football        Ball
4        Tennis          Racket
3        Ice Hockey      Puck
3        Ice Hockey      Hockey stick
```

Q: Given the sports we offer, and the equipment that's available, which sporting events could we host?
A: Football, Tennis, Ice Hockey because we both

- *Officially host the sport, and*
- *Have the relevant items in equipment*

Inner Join Syntax

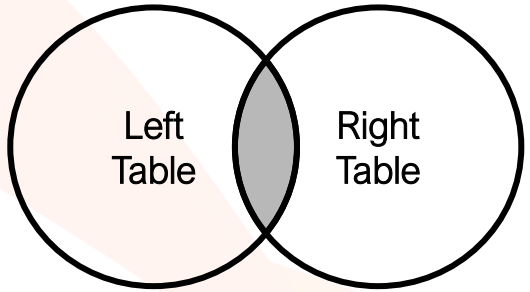


- **INNER JOIN** is the keyword used
- **JOIN** is also valid, because inner is the default join type in MySQL

```
SELECT *
FROM sports INNER JOIN equipment
ON sports.sport_id = equipment.sport_id;
```

```
-------------------------------------------------------------------
sport_id sport           team_size       item_name        sport_id
-------------------------------------------------------------------
1        Football        11      1       Ball             1
4        Tennis          1       2       Racket           4
3        Ice Hockey      18      4       Puck             3
3        Ice Hockey      18      5       Hockey stick     3
```

Inner Join Syntax



- Now that we are retrieving columns from multiple tables in one query, we must specify which **table** each **column** comes from, e.g. SELECT **equipment.item_name**
- Below are two valid ways to write an INNER JOIN to retrieve identical results.
  - Spot the difference in the queries:

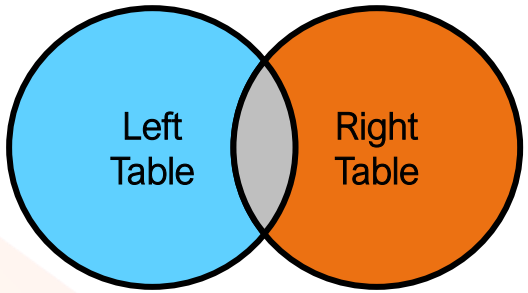
```
SELECT sports.sport_id, sports.sport,
equipment.item_name
FROM sports
INNER JOIN equipment
ON sports.sport_id = equipment.sport_id;


-------------------------------------
sport_id sport           item_name
-------------------------------------
1        Football        Ball
4        Tennis          Racket
3        Ice Hockey      Puck
3        Ice Hockey      Hockey stick
```

```
SELECT s.sport_id, s.sport,  e.item_name
FROM sports AS s
JOIN equipment AS e
ON s.sport_id = e.sport_id;


-------------------------------------
sport_id sport           item_name
-------------------------------------
1        Football        Ball
4        Tennis          Racket
3        Ice Hockey      Puck
3        Ice Hockey      Hockey stick
```

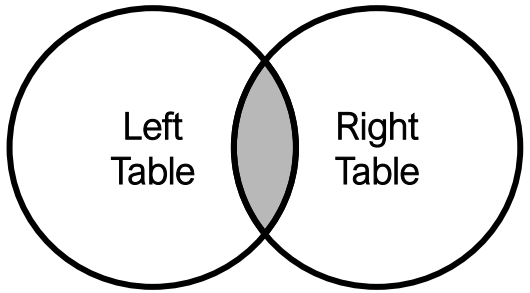
Query: Do We Have any Useless Equipment?

- So far we have only seen **symmetrical** joins (FULL JOIN, INNER JOIN) which:
  - Prioritise neither the right nor the left table.
  - Either return **all** unmatched rows (FULL) or **no** unmatched rows (INNER).

FULL JOIN

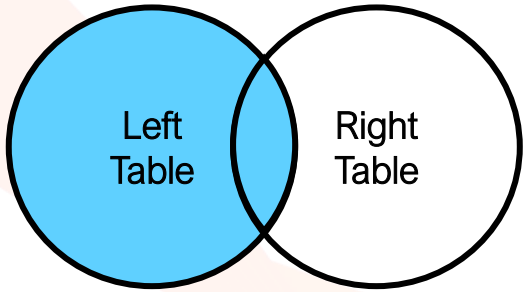


INNER JOIN



- To answer *"Do we have any useless equipment, i.e. equipment that does not match with a sport we offer"*, we could harness an asymmetrical join (LEFT JOIN or RIGHT JOIN) to return **some** mismatched rows by prioritising the left or right table.

Left Join

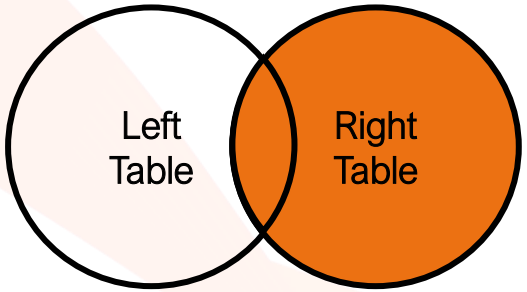


- **LEFT JOIN** returns
  - **All rows** from the **left table** and
  - The **matching rows** from the **right table**
  - If there is **no match**, **NULL** values are included

e.g. All sports that the club officially offers and any equipment that matches those sports.

```
SELECT s.sport_id, s.sport,  e.item_name
FROM sports AS s
LEFT JOIN equipment AS e
ON s.sport_id = e.sport_id;


-------------------------------------
sport_id sport           item_name
-------------------------------------
1        Football        Ball
2        Cricket         NULL
3        Ice Hockey      Hockey stick
3        Ice Hockey      Puck
4        Tennis          Racket
5        Netball         NULL
```

Right Join



- **RIGHT JOIN** returns
  - **All rows** from the **right table** and
  - The **matching rows** from the **left table**
  - If there is **no match**, **NULL** values are included

e.g. All equipment the club has, and any sports officially offered that match that equipment.

```
SELECT e.sport_id, s.sport,  e.item_name
FROM sports AS s
RIGHT JOIN equipment AS e
ON s.sport_id = e.sport_id;


-------------------------------------
sport_id sport           item_name
-------------------------------------
1        Football        Ball
4        Tennis          Racket
7        NULL            Pommel Horse
3        Ice Hockey      Puck
3        Ice Hockey      Hockey stick
```

Q: Do We Have any Useless Equipment?

- Having chosen sports to be our left table and equipment to be our right table…
- Which JOIN is best suited to answer *"Do we have any useless equipment, i.e. equipment that does not match with a sport we offer?"*

```
-- LEFT JOIN
-- All rows from sports, and any from
equipment that match on sport_id

-------------------------------------
sport_id sport           item_name
-------------------------------------
1        Football        Ball
2        Cricket         NULL
3        Ice Hockey      Hockey stick
3        Ice Hockey      Puck
4        Tennis          Racket
5        Netball         NULL
```

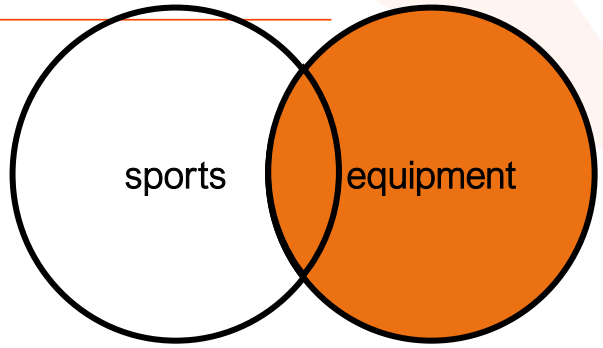
```
-- RIGHT JOIN RESULT
-- All rows from equipment, and any from
sports that match on sport_id


-------------------------------------
sport_id sport           item_name
-------------------------------------
1        Football        Ball
4        Tennis          Racket
7        NULL            Pommel Horse
3        Ice Hockey      Puck
3        Ice Hockey      Hockey stick
```

Answer: Right Join

- Having chosen sports to be our left table and equipment to be our right table, a RIGHT JOIN would suit.

- This gives **all** rows in equipment, including those which have **no match** in sports.

Unused equipment: we have a pommel horse in equipment with no matching sport in sports.

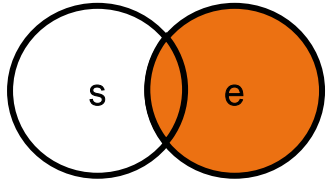



```
-- RIGHT JOIN RESULT
-- All rows from equipment, and any from
sports that match on sport_id


-------------------------------------
sport_id sport           item_name
-------------------------------------
1        Football        Ball
4        Tennis          Racket
7        NULL            Pommel Horse
3        Ice Hockey      Puck
3        Ice Hockey      Hockey stick
```

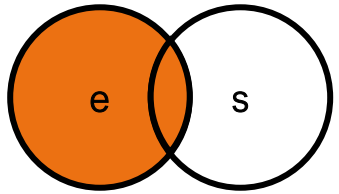

neueda

Right Joins vs. Left Joins

- You could achieve the same result using a LEFT JOIN, by switching the order of equipment and sports in the JOIN clause.




```
SELECT e.sport_id, s.sport,  e.item_name
FROM sports AS s
RIGHT JOIN equipment AS e
ON s.sport_id = e.sport_id;

-------------------------------------
sport_id sport           item_name
-------------------------------------
1        Football        Ball
4        Tennis          Racket
7        NULL            Pommel Horse
3        Ice Hockey      Puck
3        Ice Hockey      Hockey stick
```

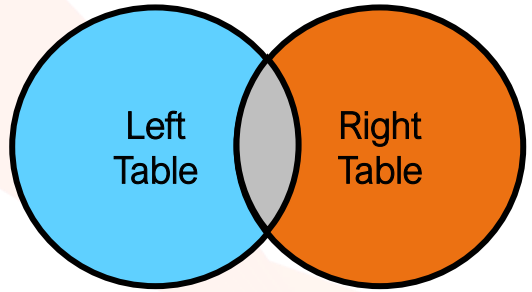



```
SELECT e.sport_id, s.sport,  e.item_name
FROM equipment AS e
LEFT JOIN sports AS s
ON s.sport_id = e.sport_id;

-------------------------------------
sport_id sport           item_name
-------------------------------------
1        Football        Ball
4        Tennis          Racket
7        NULL            Pommel Horse
3        Ice Hockey      Puck
3        Ice Hockey      Hockey stick
```

Just in Case You've Been Wondering what a Pommel Horse is This Whole Time…

https://en.wikipedia.org/wiki/Pommel_horse

Revisiting the Full Join

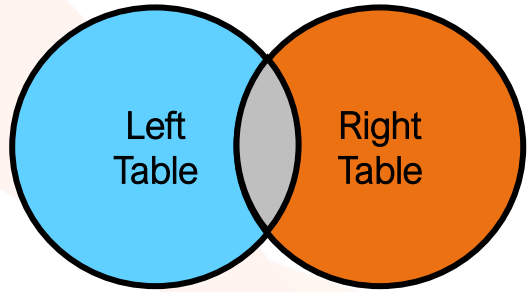


- We saw earlier that a FULL JOIN returns
  - **All rows** from both tables, **including unmatched** rows
- Lets look at the syntax for FULL JOIN.
- Unlike some other RDBMSs, MySQL does **not** have a FULL JOIN key word
  - You can create a full join in MySQL by combining LEFT JOIN, RIGHT JOIN and UNION.

```
Example FULL JOIN query result:


--columns from sports     --columns from equipment
-------------------------------------
sport_id sport            sport_id item_name
-------------------------------------
1        Football         1        Ball
2        Cricket          NULL     NULL
3        Ice Hockey       3        Hockey stick
3        Ice Hockey       3        Puck
4        Tennis           4        Racket
5        Netball          NULL     NULL
NULL     NULL             7        Pommel Horse
```


neueda

Full Join Using Left, Right, and Union




```sql
-- Full Join Using LEFT JOIN, RIGHT JOIN,
UNION

SELECT s.sport_id, s.sport,  e.item_name
FROM sports AS s
LEFT JOIN equipment AS e
ON s.sport_id = e.sport_id
UNION
SELECT s.sport_id, s.sport,  e.item_name
FROM sports AS s
RIGHT JOIN equipment AS e
ON s.sport_id = e.sport_id;
```

```
-- Query result:

-------------------------------------
sport_id sport           item_name
-------------------------------------
1        Football        Ball
2        Cricket         NULL
3        Ice Hockey      Hockey stick
3        Ice Hockey      Puck
4        Tennis          Racket
5        Netball         NULL
NULL     NULL            Pommel Horse
```

Summary

- Why joins are performed
- How rows are matched using a reference column
- Join types and their use cases:
  - INNER JOIN
  - LEFT JOIN
  - RIGHT JOIN
  - FULL JOIN (UNION)

neueda

DESIGNING & CREATING

DATABASES & TABLES

Objectives

- Entity-Relationship modelling
- Database Normalisation
- Creating a Database
- Creating a Table
- Datatypes
- Constraints
- AUTO_INCREMENT attribute

Database Design

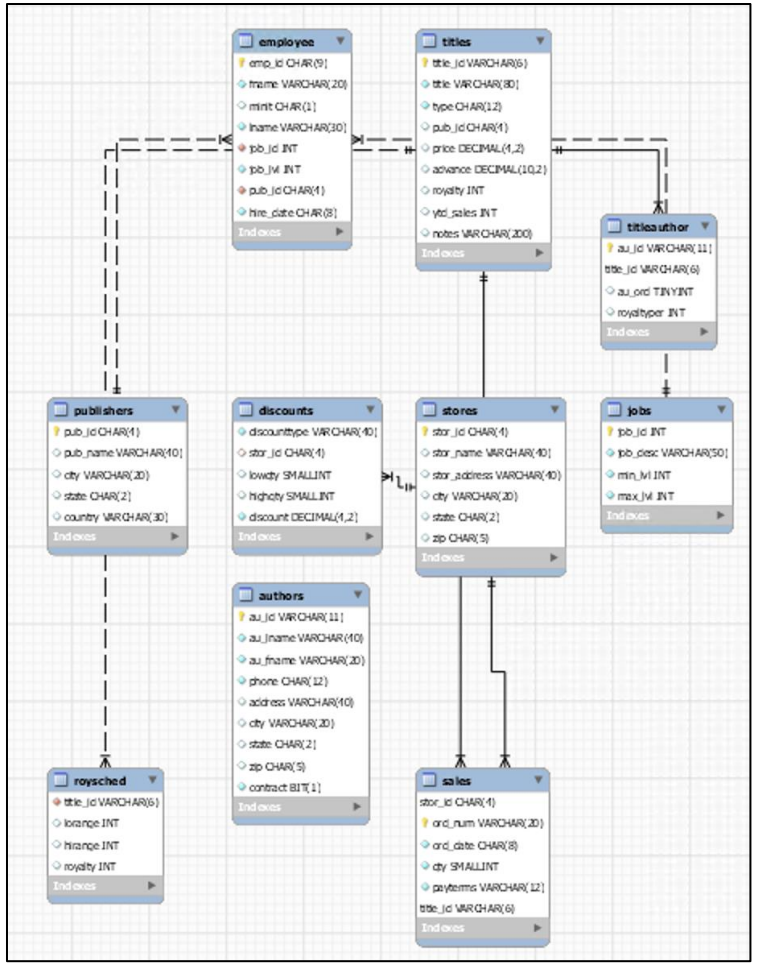
- Good **database design** ensures efficient and reliable data management
- The core principles are not complex but may take years to master
- On the following slides, you'll be introduced to the two fundamentals
  - **Entity-Relationship** (ER) **Modeling**
  - **Database Normalisation**

Entity-Relationship (ER) modelling

- **ER modelling** is a method used to conceptually structure data for creating databases

- An **ER model** represents
  - Entities
  - Attributes
  - Constraints
  - Relationships

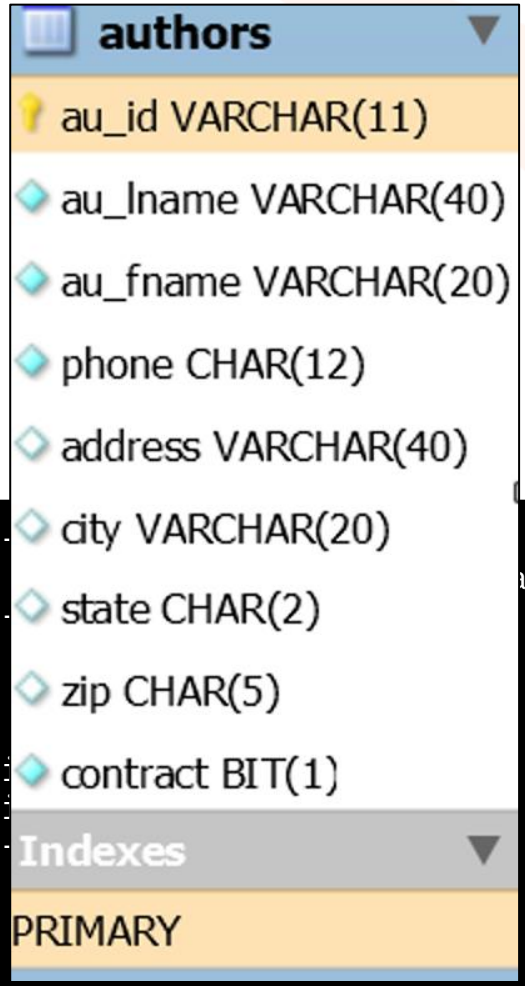
- On the following slides, you'll be introduced to the above concepts



neueda

ER modelling: Entities and Attributes

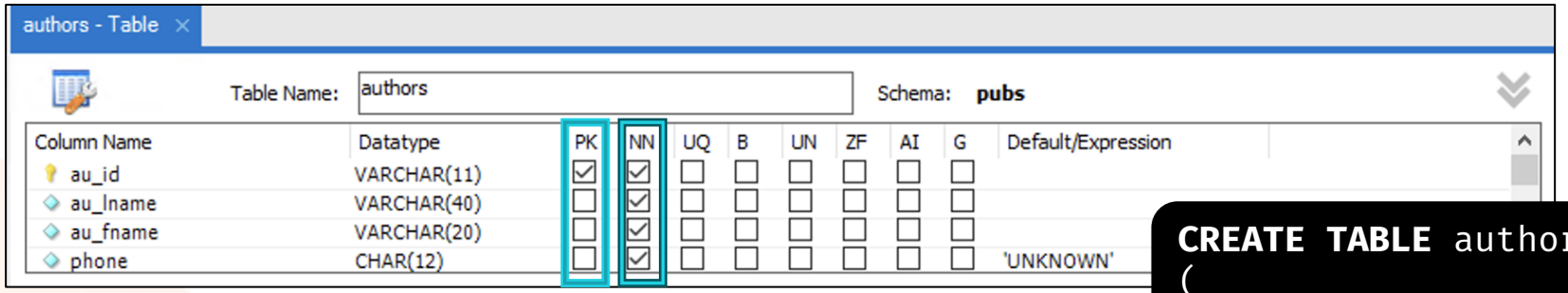
- In ER model
  - **Entities**
    - Are represented as **tables** in SQL
  - **Attributes**
    - Are represented as **columns** in the tables




```
+------------+--------------+-------------+---------------+----------------------+------------
|  au_id     |  au_lname    |  au_fname   |     phone     |       address        |      city
+------------+--------------+-------------+---------------+----------------------+------------
| 409-56-7008|  Bennet      |  Abraham    | 415 658-9932  |  6223 Bateman St.    |   Berkeley
| 213-46-8915|  Green       |  Marjorie   | 415 986-7020  | 309 63rd St. #411    |   Oakland
| 238-95-7766|  Carson      |  Cheryl     | 415 548-7723  |  589 Darwin Ln.      |   Berkeley
| 998-72-3567|  Ringer      |  Albert     | 801 826-0752  |    67 Seventh Av.    | Salt Lake C
| 899-46-2035|  Ringer      |  Anne       | 801 826-0752  |    67 Seventh Av.    | Salt Lake C
+------------+--------------+-------------+---------------+----------------------+------------
Note:
This is a partial data representation
of the "authors" table
```

ER modelling: Constraints

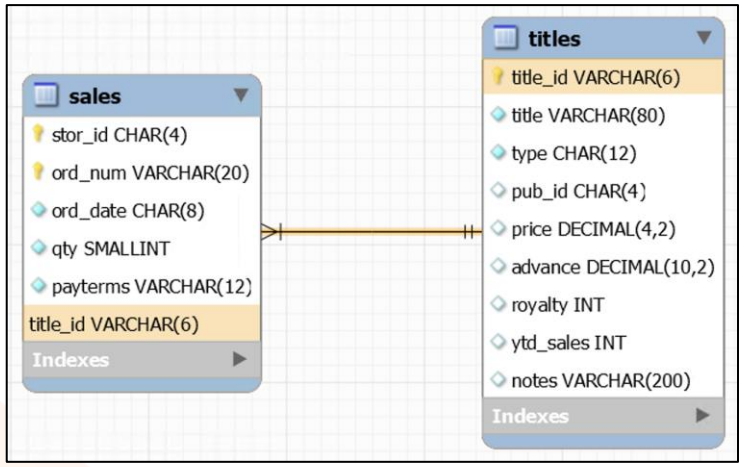
- In ER model **Constraints**
  - Translate into constraints in SQL tables
    - Constraints help enforce data integrity in a database
    - Examples: Primary Key, Not Null
      - You'll be provided with more explanations later on




```
CREATE TABLE authors
(
    au_id varchar(11) NOT NULL,
    au_lname varchar(40) NOT NULL,
    au_fname varchar(20) NOT NULL,
    phone char(12) NOT NULL DEFAULT 'UNKNOWN',
    PRIMARY KEY(au_id)
);
```

ER modelling: Relationships

- **Relationships** in an ER model
  - signify the **association between two entities**




```
-- This is the sales table
-- It includes a foreign key reference to the titles table

CREATE TABLE sales
(
    stor_id char(4) NOT NULL,
    ord_num varchar(20) NOT NULL,
    ord_date char(8) NOT NULL,
    qty smallint NOT NULL,
    payterms varchar(12) NOT NULL,
    title_id varchar(6) NOT NULL,
    FOREIGN KEY(title_id) REFERENCES titles(title_id),
    PRIMARY KEY(stor_id, ord_num, title_id)
);
```

Database Normalisation

- **Database Normalisation** is a strategy used to
  - Design relational databases efficiently
- By breaking down tables and removing duplicates
  - Eliminates redundant data and
  - Ensures logical data dependencies
- In some cases, it may be advantageous to **denormalise**
  - Certain parts of your database
  - For performance reason
- A simple example of database normalisation on the following slides

Database Normalisation: 1NF, 2NF, 3NF

- **First Normal Form (1NF)**
  - One slot in a bookshelf holds one book, not multiple
  - One cell in a table holds one value, not multiple
- **Second Normal Form (2NF)**
  - Each slot should only contain information relevant to the book in it
  - Each table should only contain data related to its primary key
- **Third Normal Form (3NF)**
  - We don't store the same friend's address in different slots
  - All non-key information in a table should directly depend on the primary key
    - To avoid duplication and inconsistencies

Database Normalisation: Before

- We start with a single table that contains all the data

stor_id	stor_addr	ord_num	qty	payterm	title_id	title	discount
S1	Addr1	O1	5	Net 30	T1	Book1	5%
S1	Addr1	O2	3	Net 60	T2	Book2	5%
S2	Addr2	O3	7	Net 30	T3	Book3	10%
S2	Addr2	O4	2	Net 60	T1	Book1	10%

- By breaking down our initial unnormalised table into four tables
- You can adhere to the principles of 1NF, 2NF, and 3NF
  - Reducing data redundancy and improving the integrity and flexibility
- See the tables on the following slide

Database Normalisation: After

Table: Stores

stor_id (PK)	stor_address
S1	Address1
S2	Address2

Table: Discounts

discounttype	stor_id (FK)	discount
Type1	S1	0.10
Type2	S2	0.20

Table: Sales

stor_id (FK)	ord_num	qty	payterms	title_id (FK)
S1	O1	100	Term1	T1
S2	O2	200	Term2	T2

Table: Titles

title_id (PK)	title
T1	Title1
T2	Title2

- Discounts and Sales have Foreign Keys (FK) to Stores table
- Sales table has a Foreign Key (FK) to Titles table

Creating Databases and Tables

- Having ER modelling and Database Normalisation in mind, we can start creating databases and tables

- On the following slides, you'll go through the following topics
  - **CREATE TABLE** syntax
  - **Datatypes**
  - **Constraints**
  - **AUTO_INCREMENT** attribute

Creating a Table: the syntax

- Having ER modelling and Database Normalisation in mind
  - We can start creating databases and tables

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

- **CREATE TABLE** is the statement used to create a table
- **table_name** is the table's name.
- **column1**, **2**, **3** are the names of the columns.
- **datatype** specifies the type of data the column can hold (e.g., INT)

Creating a Table: the Data Types

- **MySQL** supports a variety of **Data Types**
  - **Numeric** Types:
    - INT, TINYINT, SMALLINT, MEDIUMINT, BIGINT, FLOAT, DOUBLE, DECIMAL
  - **Date** and **Time** Types
    - DATE, DATETIME, TIMESTAMP, TIME, YEAR
  - **String** Types
    - CHAR, VARCHAR, BINARY, VARBINARY, TEXT, BLOB, ENUM
  - **Special** Types
    - GEOMETRY, POINT, LINESTRING, POLYGON
  - **JSON** Type
- Check with the documentation on which to use and how to customise them

Creating a Table: the Constraints 1 of 3

- Table constraints in MySQL allow you to define rules
  - On the data that is being inserted into the table
  - This can help ensure data integrity and accuracy

- Here are some common MySQL table constraints:
  - **NOT NULL**
  - **UNIQUE**
  - **PRIMARY KEY**
  - **FOREIGN KEY**
  - **CHECK**
  - **DEFAULT**

- Some examples on the following slides

Creating a Table: the Constraints 2 of 3

- **NOT NULL** ensures that a column cannot have a NULL value

```
CREATE TABLE Employees (
    ID INT NOT NULL,
    FirstName VARCHAR(100) NOT NULL
);
```

- **UNIQUE** ensures that all values in a column are unique.

```
CREATE TABLE Employees (
    ID INT NOT NULL UNIQUE,
    FirstName VARCHAR(100) NOT NULL
);
```

Creating a Table: the Constraints 2 of 3

- **PRIMARY KEY** is a combination of NOT NULL and UNIQUE
  - A table can have only one primary key
  - It uniquely identifies each record in a table

- **FOREIGN KEY** It is a column or a set of columns that is used to
    - Establish and enforce a link between the data in two tables
    - Prevent actions that would destroy links between tables

```
CREATE TABLE Orders (
    OrderID INT NOT NULL,
    OrderNumber INT NOT NULL,
    EmployeeID INT,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (EmployeeID) REFERENCES Employees(ID)
);
```

Creating a Table: the Constraints 2 of 3

- **DEFAULT**
  - Is used to provide a default value for a column
- **CHECK**
  - Is used to limit the value range that can be placed in a column

```
CREATE TABLE Employees (
   ID INT NOT NULL,
   Age INT,
   Country VARCHAR(50) DEFAULT 'USA',
   CHECK (Age >= 18),
   PRIMARY KEY (ID),
);
```

AUTO_INCREMENT attribute

- **AUTO_INCREMENT (**often used with a primary key)
  - Automatically generates a unique number for a column
    - Each time a new record is inserted into a table

```
CREATE TABLE Customers (
    ID INT AUTO_INCREMENT,
    FirstName VARCHAR(100),
    LastName VARCHAR(100),
    PRIMARY KEY(ID)
);

INSERT INTO Customers (FirstName, LastName) VALUES ('John', 'Doe');
```

- MySQL will automatically assign a value for the ID
  - The first record you insert will get an ID of 1, the next an ID of 2 and so on
  - AUTO_INCREMENT does not reset or reuse the ID from the deleted row

Summary

- Entity-Relationship modelling
- Database Normalisation
- Creating a Database
- Creating a Table
- Datatypes
- Constraints
- AUTO_INCREMENT attribute

neueda

SQL Functions

Objectives

- SQL Functions
- String Functions
- Date and Time Functions
- Aggregate Functions
- Control Flow Functions

SQL Functions

- SQL functions are built-in commands that perform operations on data
- They can be used in SELECT, WHERE, and ORDER BY clauses of an SQL
- MySQL supports a wide range of functions
  - Some functions are specific to MySQL
  - Always check the documentation of a specific DBMS
- Some examples on the following slides

String Functions

- E.g. **CONCAT**, LIKE, REPLACE, UPPER

```
-- Combining the author's first and last name
SELECT CONCAT(au_fname, ' ', au_lname) as 'Full Name' FROM authors;


+------------------------+
|       Full Name        |
+------------------------+
| Abraham Bennet |
| Marjorie Green     |
| Cheryl Carson      |
+------------------------+
```

Date and Time Functions

- E.g. NOW, DATE, **YEAR**, DAY, HOUR, SYSDATE

```
-- Extract Date from employee
SELECT YEAR(hire_date) AS 'Hire Since' FROM employee WHERE emp_id='A-C71970F';


+-----------+
| Hire Date |
+-----------+
|   1991    |
+-----------+
```

Aggregate Functions

- E.g. AVG, **COUNT**, MAX, MIN, SUM

```
-- Counting the number of authors
SELECT COUNT(*) FROM authors;
+-------+
| COUNT |
+-------+
|   3   |
+-------+
```

Control Flow Functions

- **E.g. CASE**, IF, LOOP

```
-- Assigning a label based on the price of the book
SELECT title, price,
CASE
    WHEN price < 10 THEN 'Cheap'
    WHEN price >= 10 AND price < 20 THEN 'Affordable'
    ELSE 'Expensive'
END as 'price_category'
FROM titles;
```

```
+----------------------------------+-------+----------------+
| title                            | price | price_category |
+----------------------------------+-------+----------------+
| Secrets of Silicon Valley        | 20.00 | Expensive      |
| The Busy Executive's Database... | 19.99 | Affordable     |
| Emotional Security: A New Algo.. | 7.99  | Cheap          |
+----------------------------------+-------+----------------+
```

Basic Arithmetic Operations

- In MySQL, you can use, e.g. *** operator** to perform multiplication
- This is a fundamental operation and not technically a function
- The same with the other operators: **+**, **-**, **/** and **%**

```
-- Calculating the discounted price after applying a 10% discount
SELECT title, price, price * 0.9 as 'discounted_price' FROM titles;


+----------------------------------+-------+------------------+
| title                            | price | discounted_price |
+----------------------------------+-------+------------------+
| Secrets of Silicon Valley        | 20.00 | 18.00            |
| The Busy Executive's Database... | 19.99 | 17.99            |
| Emotional Security: A New Algo.. | 7.99  | 7.19             |
+----------------------------------+-------+------------------+
```

Summary

- SQL Functions
- String Functions
- Date and Time Functions
- Aggregate Functions
- Control Flow Functions