



Institute_{of}
Data

Software Engineering

Module 8

Databases



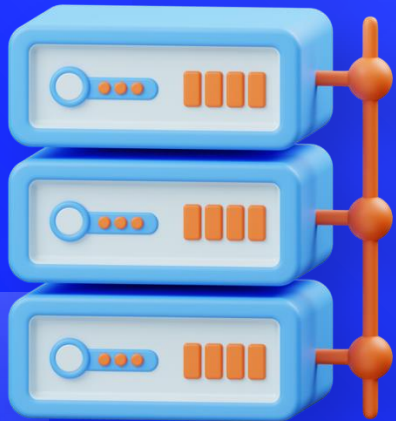
Agenda

- Section 1 Introduction to Databases
- Section 2 Database Design
- Section 3 Introduction to MongoDB
- Section 4 Introduction to MySQL
- Section 5 Introduction to Redis



Section 1:

Introduction to Databases



In computing, a database is an **organized collection of data** stored and accessed **electronically**.

Small databases can be stored on a standard file system, while large databases are typically hosted on computer clusters or cloud storage.

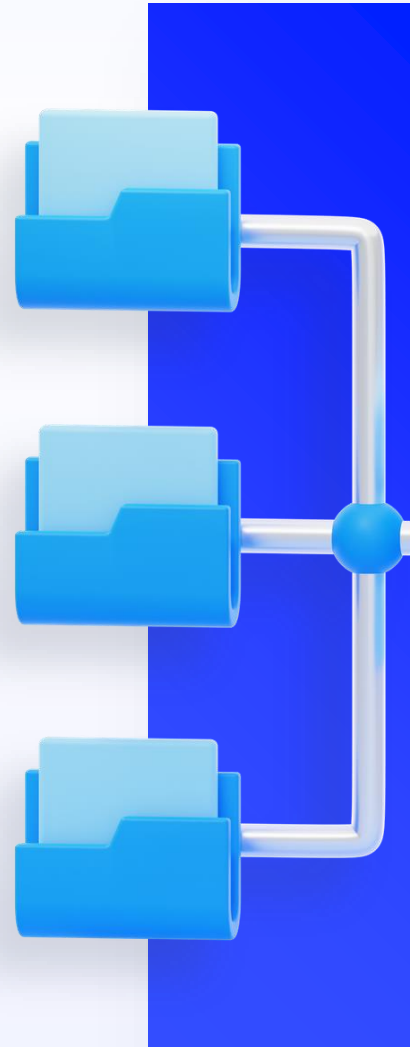
What is a Database ?

It's a **collection of data**, stored in an **organised** method that enables **easy access** and **management**.

Each database has different rules applied to it, and these rules are configurable depending on the type of data and what kinds of issues are relevant to it. Perhaps your issue is one of size, whereas someone else has a smaller amount of data with a high level of sensitivity.

A database handles things you can't see that are going on in the background: data security, enforced integrity, the ability to access data quickly and reliably, and robustness; serving a large number of users at once and even surviving crashes and hardware issues without data corruption.

When designing and setting up a database, we need to consider the underlying structure of the data as well as these issues, to ensure it is managed efficiently and reliably.

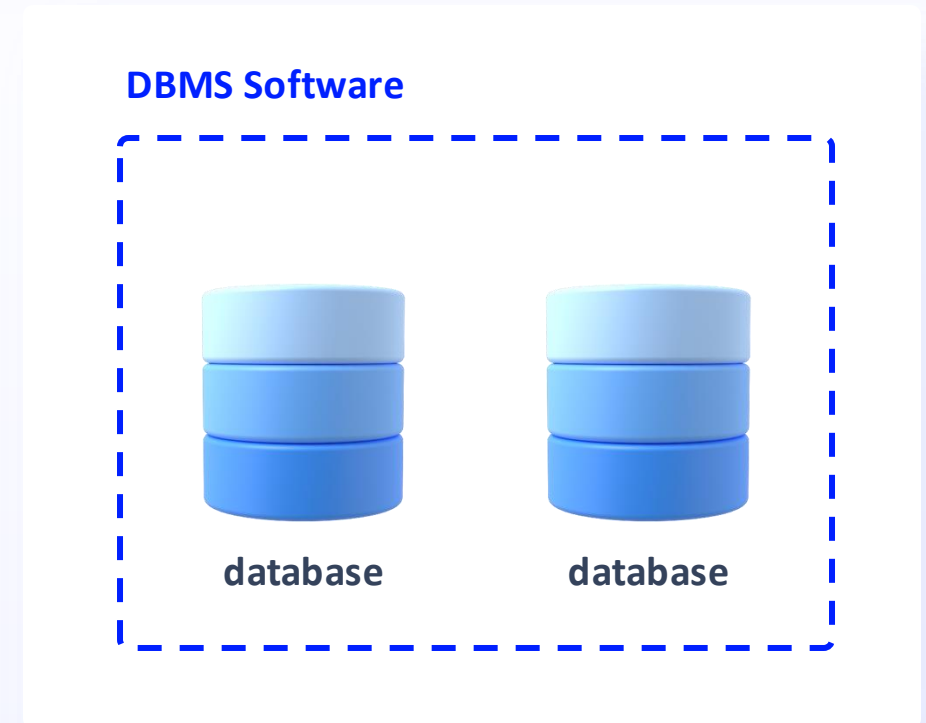


Database Management Systems (DBMS)

We frequently make the mistake of referring to our database as **Oracle**, **MySQL**, **SQL Server**, or **MongoDB**. However, these are **database management systems** (DBMS), not databases.

The database management system (**DBMS**) is **software** that you install on your computer or on a server, and then use to **administer** one or more databases.

The **database** contains your actual data as well as the rules that govern it, whereas the database management system (**DBMS**) is the **program** that surrounds and manages potentially multiple databases, while also enforcing the rules you set for them. The type of data, such as integer or string, or the relationship between them, are examples of database rules.



Typically databases use the above cylinder shape when represented on a diagram.

Database Management Systems (DBMS)

There are many different types of DBMS, and each have their own strengths and weaknesses. The right choice depends on the **type of data** being stored, as well as its **management focus areas** (such as speed and security) and **non-functional factors** such as developer experience, cost, vendor and company preference. It is not uncommon to use several different databases and different DBMSes within the same company.

DBMSes can be categorised into several major categories, each with various popular examples:

❖ Relational Database Management Systems

RDBMSes are the most popular and **widely-used**, having been around since the 70s and matured and improved continually since then. Examples include **Oracle**, **SQL Server** and **MySQL**. Data is structured into **rows** and **columns** inside **tables**, with **relationships** between tables, and firm rules about how data is stored. They use a **Structured Query Language** (SQL) to access and manipulate the data efficiently.

Database Management Systems (DBMS)

❖ NoSQL Database Systems

In contrast to a **relational** DBMS, a NoSQL DBMS doesn't require the data to follow a pre-determined structure or set rules about data types, giving it more **flexibility** and better ability to cope with **high-volume, high-velocity** workloads of **real-time** data sourced from the Internet. Examples include **Cassandra** and **MongoDB**, with data stored as **collections** of document **objects**.

Sub-types such as key-value, column-family, graph and time-series databases are also NoSQL.

❖ Hierarchical Database Systems

A **hierarchical** DBMS stores data in a hierarchical, **tree-like** structure where each **record** has a single **parent**. Originally the precursor to RDBMSes, they were largely replaced by relational databases as these offer more flexible relationships. Today they are still used, primarily for storing **geographical** or banking information, in examples such as **IMS** or the **Windows Registry**.

Database Management Systems (DBMS)

❖ Object-Oriented Database Systems

OODBMS offer complex **object modelling** but lack standardisation and may suffer from performance and scalability issues. Their maturity and adoption are lower than relational databases, leading to higher costs, **integration challenges** and **learning curves**. Despite these drawbacks, OODBMS excel in applications with intricate object relationships and tight integration with object-oriented programming, since data is stored in objects. Examples include **db4o**, **ObjectDB** and **Versant**.

❖ Network Database Systems

In a Network Database Management System (NDBMS), data is stored in a **network-like structure**, with **records** linked via **pointers**. They are not widely used today due to complexities, limitations and more efficient options. However, some **legacy** systems persist in specific domains like banking and telecommunications. Examples include **IDMS**, **IMS** and **Raima**.

Database Management Systems (DBMS)

Selecting the right DBMS is crucial for the success of any project. To make a well-informed decision:

1. Understand Your Data Requirements:

- Assess your data needs, including volume, variety, velocity, and value.
- Find the data model that best fits your data structure: relational, document-oriented, key-value, etc.

2. Evaluate Performance and Scalability:

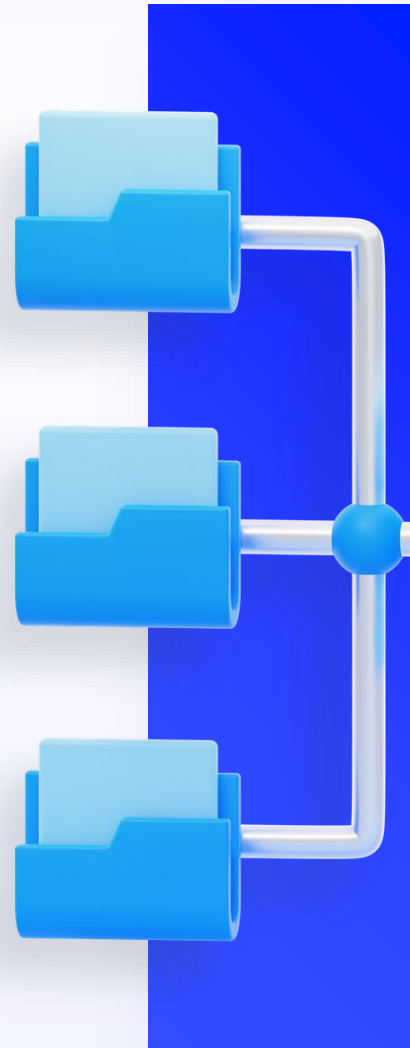
- Analyse performance benchmarks and scalability capabilities of different DBMS options.
- Determine if the DBMS can handle your anticipated workload without sacrificing performance.

3. Consider Data Consistency and Integrity:

- Assess your requirements for data consistency, transactional integrity, and ACID compliance.
- Evaluate the DBMS's support for data replication, backup, and disaster recovery.

4. Factor in Cost and Resources:

- Consider both upfront and ongoing costs, including licensing, hardware, and maintenance expenses.
- Assess the availability of skilled personnel and community support for the chosen DBMS.
- Evaluate vendor reputation, reliability, and long-term viability.



ACID Properties for Transactions

Since data integrity is a vital aspect of a good database, any good DBMS will guarantee that a database is in a consistent state after running each transaction.

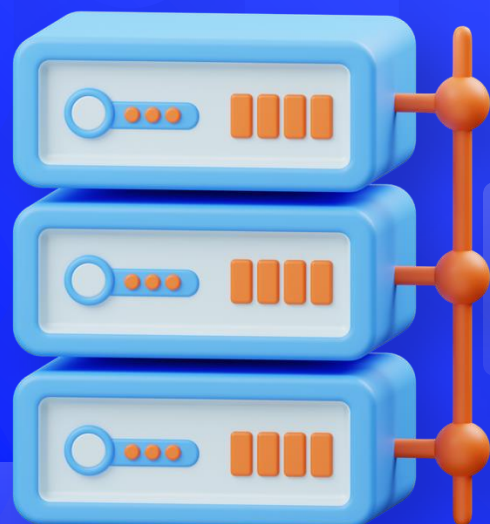
A **transaction** comprises database operations that must all succeed or fail together, and can affect one or more records. For instance, imagine a money transfer function affecting multiple account records. If money is deducted from the source but not credited to the destination, or vice versa, the system faces a consistency issue. Both operations must succeed or fail together to maintain consistency.

ACID stands for **A**tomicity, **C**onsistency, **I**solation, and **D**urability.

These properties collectively guarantee that data in a database will always be in a valid state, even in the face of unforeseen errors. Partially executed transactions will be reverted, data will always conform to its specified constraints, each transaction runs separately to others affecting the same data, and unexpected hardware/software failures will not cause inconsistent data.

Section 2:

Database Design



Database design is a collection of actions or processes that help with the **planning, development, deployment, and maintenance** of enterprise data management systems. A well-designed database reduces the time and cost of **accessing** and **maintaining** data, and helps to ensure data **consistency**. The designer must adhere to the limitations of the chosen DBMS and determine **what** data must be saved, the best way to represent and enforce its inherent structure, and how the elements **relate** to one another.

The primary goals of database design are to create **physical** and **logical models** of the database system under consideration. To clarify, the **logical** model focuses primarily on **what data is stored**, and serves as a blueprint for understanding the business domain through entities, attributes, relationships, and constraints in a language-agnostic and platform-independent manner. The **physical** database design model, on the other hand, specifies **how** the data from the logical design model is physically stored, using hardware resources and storage structures within a specific Database Management System (DBMS).

Importance of Database Design

Database design is a crucial task when building a back-end system. If done well, it should:

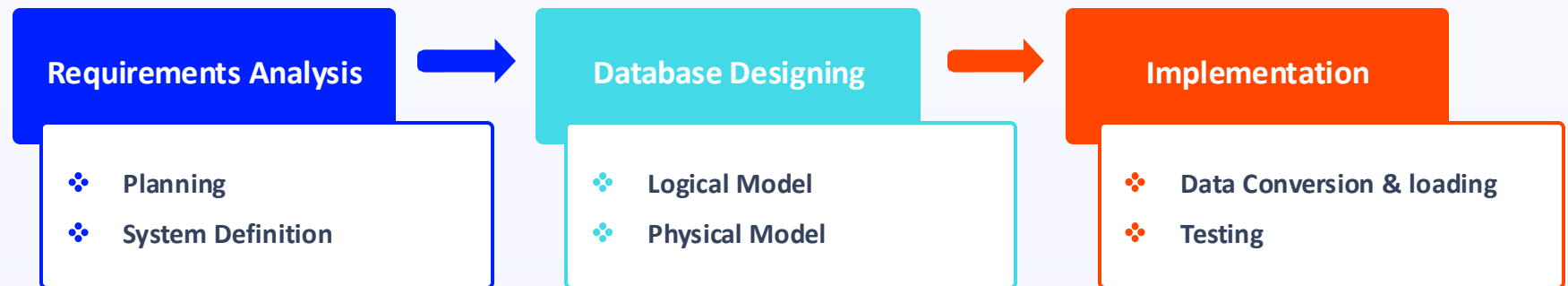
- ❖ **Accurately Capture User Requirements:** Thoroughly understanding and incorporating all user requirements into the database design ensures that the system meets business needs effectively, minimizing the risk of future rework or dissatisfaction.
- ❖ **Ensure Data Integrity:** By identifying data constraints, it maintains accuracy and consistency, preventing anomalies like duplicate records.
- ❖ **Enhance Performance:** Data optimisation strategies improve speed and scalability, ensuring efficient query execution, especially with growing data volumes.
- ❖ **Facilitate Maintenance:** Clear schema structures ease modifications and additions, ensuring adaptability and reliability over time.



Database Design Lifecycle

The database design lifecycle contains three major steps:

- ❖ Requirements Analysis
- ❖ Database Designing
- ❖ Implementation



All three steps are vital to achieving an accurate and efficient data storage model for an application.

Database Design Lifecycle: Requirements Analysis

First and foremost, a strategy must be developed to determine what the project's essential **requirements** are and how to best analyse and capture these, so that the database design can proceed based on accurate and complete information.

Planning - This stage is involved with the overall DDLC planning (Database Development Life Cycle). In this initial stage, strategic considerations such as timing and cost are taken into account.

System Definition - After planning, this step covers the boundaries and scopes of the correct database. It details what will and will not be included in the database, but does not yet attempt to describe a structure for this data.

Database Design Lifecycle: Database Designing

The next phase is to **design** the database around the user's requirements, using different models to avoid imposing too much burden or significant reliance on a single feature. As a result, a model-centric approach has emerged, with logical and physical models playing a critical role.

Logical Model - The primary goal of this stage is to create a model based on the specified requirements, which identifies each entity within the data, as well as the attributes and relationships for these entities. The model is created on paper (or a drawing tool) with no implementation or database management system considerations.

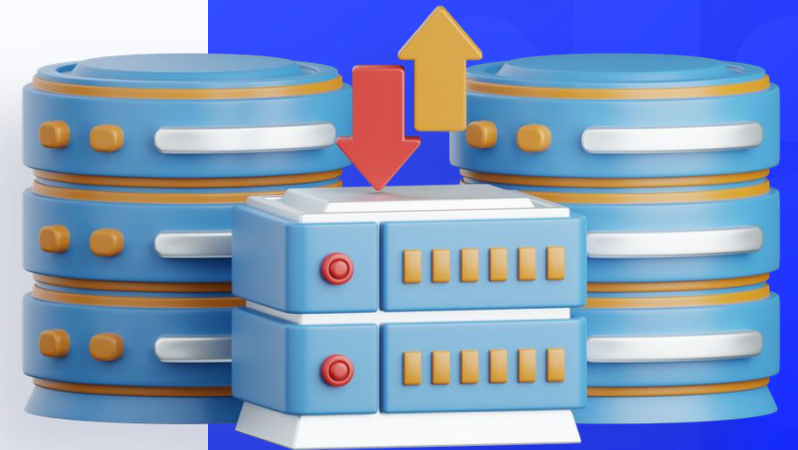
Physical Model - The physical model is concerned with the logical model's practices and implementations. It considers aspects such as data types, speed, security and volume, and involves choosing an appropriate DBMS.

Database Design Lifecycle: Implementation

The next step is to look at the **implementation** techniques and ensure they satisfy our criteria. Continuous **integration testing** of the database with various **datasets** and **data conversion** into machine understandable language are used to ensure this. Data manipulation is only introduced in these phases, which include running queries against benchmarks and determining whether or not the database is constructed satisfactorily.

Data Conversion and loading - This stage involves importing and converting data from any old system/s into the new.

Testing - This stage focuses on identifying errors in the newly implemented system. Testing is critical since it directly examines the database and compares the requirement specifications.



Important Terms

When talking about database design there are a few terminologies that are very important to understand.

- ❖ **Entity** - A single unique object, thing or concept in the real world about which information is stored. For example, in an ecommerce system entities may include customers, products, and orders.
- ❖ **Attribute** - A characteristic or trait of an entity that describes it in some way. For example, a customer has a name, an address and contact details - these are all attributes of that entity.
- ❖ **Relationship** - The link between two entities. For example, a customer may place many orders. Relationships can typically be one-to-one, one-to-many, or many-to-many.
- ❖ **Primary Key** - A unique identifier for each entity, used to ensure that each entity value is distinct and can be easily located.
- ❖ **Foreign Key** - A link to another entity's Primary Key, used to establish relationships.
- ❖ **Normalisation** - The process of structuring entities, attributes and relationships to reduce redundancy and dependency, ensuring efficiency and data integrity.

Normalisation

Normalising a database design is all about **eliminating redundancy**, so that data is never duplicated or stored in multiple places, and ensuring **consistent dependencies**, so that data for each entity is self-contained, and uses foreign keys to maintain any links to other entities. It takes place after creating an initial logical model to ensure that the final data model is as efficient as possible.

There are technical terms for the process, which is about adhering to a series of 'normal forms', but at its core we should be asking the following questions:

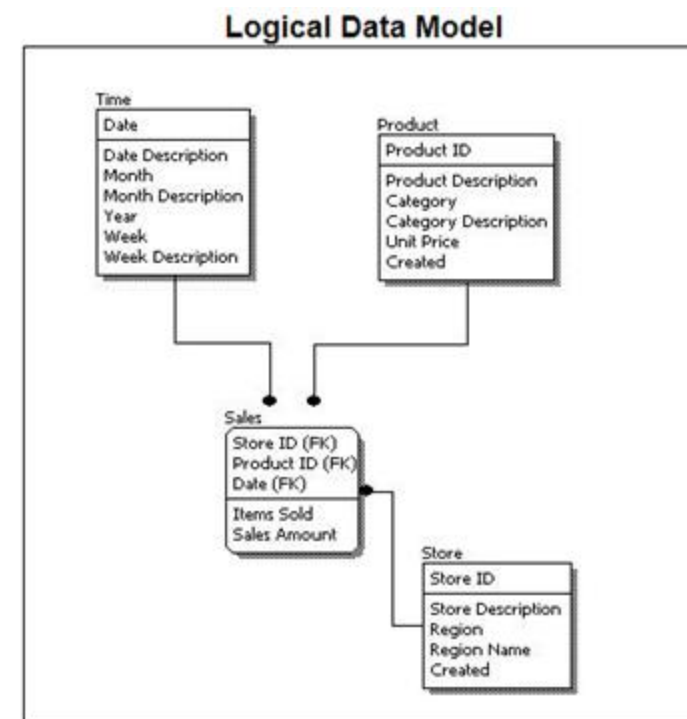
1. Does each entity have a **primary key** - a unique, stable way to reference that entity individually?
2. Are there any '**repeating groups**' inside an entity? If so, this generally indicates the need for two separate entities with a one-to-many relationship.
3. Does each attribute of an entity **directly relate** to that entity specifically? If not, you may need to move it into another, more relevant entity or create a new entity.
4. Can any attributes be **calculated** from other data? If you have attributes such as total, age, or average which are directly dependent on values in other attributes that may change over time, it is redundant to store them independently. Instead, calculate those values only when needed.

Data Models: Logical Model

A logical data model explains the data and captures its structure in as much detail as possible without having to worry about the database's physical implementations.

The following are some features of a logical data model:

- ❖ All of the entities and their **relationships**.
- ❖ Each entity has well-defined **attributes**.
- ❖ The **primary key** for each entity is specified.
- ❖ Foreign **keys** are defined as identifiers for relationships between various entities.
- ❖ At this stage, **normalisation** takes place.



Data Models: Logical Model

A logical model can be designed using the following approach:

- ❖ Make a list of all the unique **entities**, and identify or assign a **primary key** for each one.
- ❖ Determine the properties (**attributes**) of each entity.
- ❖ Concurrent **relationships** between distinct entities should be specified and identified as being **one-to-one**, **one-to-many** (ideal), or **many-to-many**.
- ❖ Many-to-many relationships must be **resolved**, usually by creating a third linking entity.
- ❖ Carry out the **normalising** procedure.

It is crucial to critically **review** the design based on the initial **requirement analysis** after carrying out this approach. If the above stages are rigorously followed, there is a high probability of constructing a highly efficient database architecture.



Data Models: Relationships

Once we have identified multiple separate entities with primary keys as part of our data model, we can start to specify the **relationships** between these entities. This is where **foreign keys** come into play.

The goal for our model is to **eliminate redundancy** and keep entities **self-contained**. So, if one customer places **multiple** orders (one-to-many relationship), we want to keep **customer** and **order** data separate, but we still need to capture the relationship. By storing the customer ID (primary key) within the orders entity as a foreign key, it keeps track of which customer has placed each order without duplicating customer data.

An example of a one-to-one relationship would be a database tracking people and passports. Each person can only have one passport, and vice versa, so those two tables would have a one-to-one relationship.

The third option is known as a **many-to-many** relationship. In a database tracking authors and books, each author can write **multiple** books and each book can have **multiple** authors. By defining a many-to-many relationship in a **linking entity** via foreign keys, we can link these two tables together.

Class Exercise: Soccer Club Database

Your local soccer club is in need of a database to store and manage their teams and player registrations.

It needs to keep track of each player's name, age, team, parental contact and payment status so they can easily see who has paid for each season, if they're playing in the right division for their age, check which team uniform they need, and contact parents when needed. Players may belong to multiple teams if they are 'playing up' an age division. Divisions are u10s, u12s, u14s, u16s, u18s and open, and are mixed genders. Teams should have around 13-20 players each, and each team has a head and assistant coach. Coaches are volunteers and rotate frequently due to changing commitments, so keeping contact details for previous coaches is handy.

Using this scenario, collaborate as a class to create a logical diagram by following the process on the previous slides:

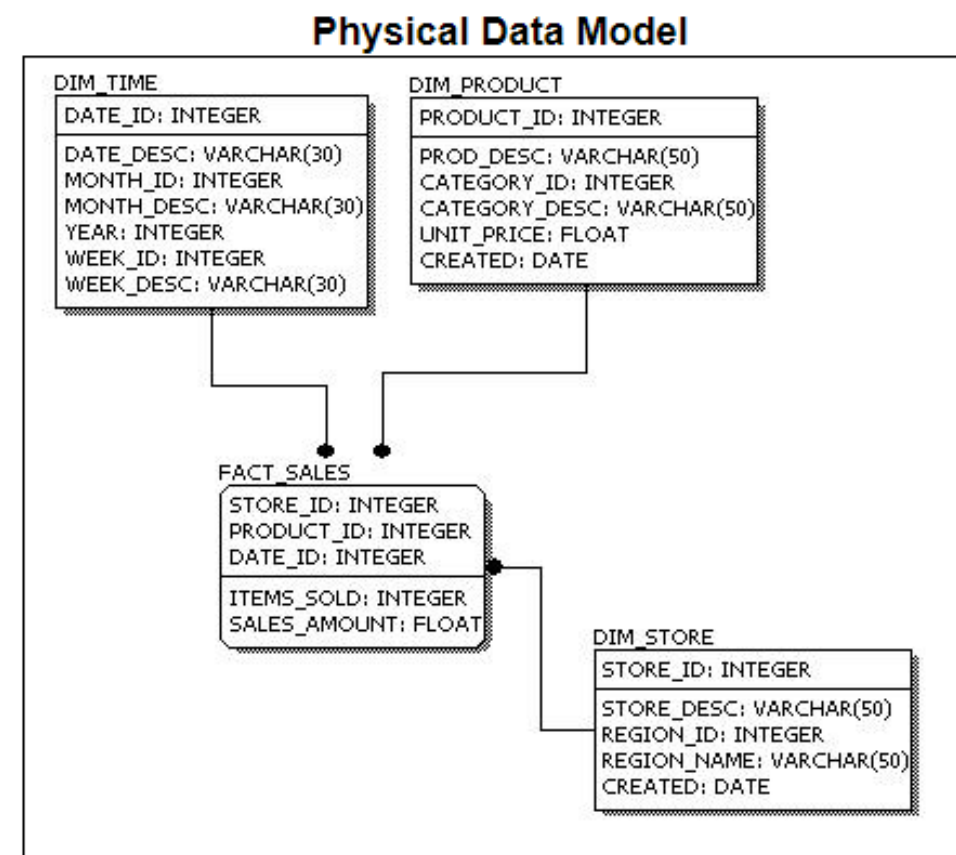
1. Identify entities & attributes.
2. Include primary keys.
3. Represent relationships using foreign keys.
4. Normalise to remove redundancy.

A popular tool for creating database diagrams is [app.diagrams](https://app.diagrams.net/)- use the Entity Relation shapes.

Data Models: Physical Model

A physical data model describes how the data will be implemented as a database in a particular DBMS. It includes details such as tables, columns, data types, indexes, and storage structures. The following are some characteristics of a physical data model:

- ❖ All tables (entities) and columns (attributes) are named.
- ❖ Data types/constraints for each attribute are chosen, using DBMS-specific syntax and features for implementation.
- ❖ Strategies for scalability, such as partitioning and indexing may be considered and specified.
- ❖ Distinct DBMSes may require different physical models.



Data Models: Physical Model

While designing a physical data model, the following points should be taken into consideration:

- ❖ Select the **DBMS** that best suits your project requirements, considering factors like scalability, performance, and compatibility with existing systems.
- ❖ Translate entities and attributes from the logical data model into **tables** and **columns** (for an RDBMS).
- ❖ Specify **data types**, **lengths**, and **constraints** for each column based on the requirements and characteristics of the chosen DBMS.
- ❖ Identify columns that will be frequently used in search criteria or join operations and create indexes on them to improve query performance.
- ❖ Fine-tune the physical data model for **performance** by denormalizing or partitioning tables, as needed.
- ❖ Evaluate **storage allocation**, **data compression**, and **caching** mechanisms to optimize resource usage and query response times.

Points 4-6 are outside the scope of this course material, but are part of real-world database architecture.

Exercise 1

Design a Logical and Physical Model for a general Blogging application.

Requirements:

- ❖ The system should store users (using basic user-related fields)
- ❖ The users should be able to create multiple posts (posts should be very basic with title, description and image)
- ❖ Other users should be able to like the posts and comment on the posts.

Note: This exercise will feed into several subsequent exercises in Modules 8 and 9, so it is important to spend time carefully following the process and review your answer with your trainer/s.

Section 3:

Introduction to MongoDB



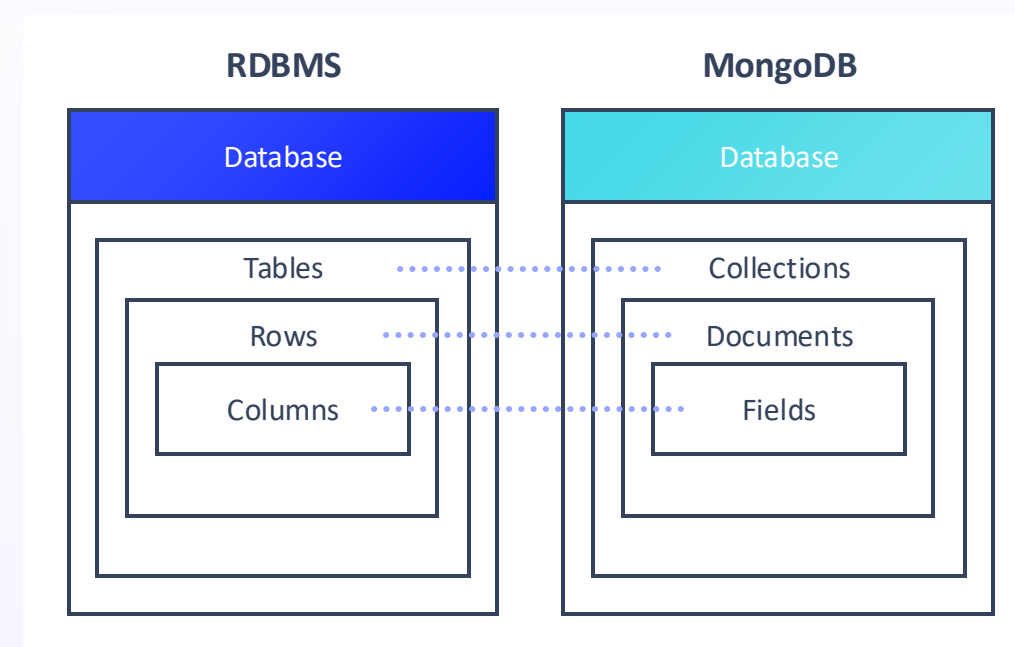
MongoDB is a document-oriented NoSQL database for storing large amounts of data. Its data model permits hierarchical connections to be represented.

Instead of using tables, rows and columns as in standard relational databases, it employs JSON-like objects (called documents) with optional schema. The basic units of data in MongoDB are documents that include key-value pairs.

MongoDB - Concepts

Each MongoDB **document** can have a number of **fields** (values) that contain information about the record. Because two separate documents can store a different amount of fields, one document's schema and size may differ from another inside the same entity (collection).

Every **field** has a key (name) that identifies it. Any **data type**, such as common text, number, and so on, can be used in the field value. A field can also be an array or a link to another document. MongoDB employs BSON (a binary encoding variant of JSON) to incorporate data types that aren't compatible with JSON, such as dates.



Although no required relationship exists between the fields, linkages and data models can be formed throughout the creation process. Data models make it easier to store arrays, construct hierarchical relationships, and so forth.

Unlike an RDBMS, data is stored in a stream rather than a schema. The Mongo DBMS does not enforce any schema.

The Mongo shell, a JavaScript interface provided by MongoDB, is used to query and create documents.

MongoDB - Features

MongoDB's entry to the market has resulted in an increase in the number of developers adopting it in specific applications due to its numerous perks and advantages, including popular apps like Uber, Twitch and Adobe.

Scalability - MongoDB is a Big Data database designed specifically for processing large amounts of data. It can store massive amounts of data, making it extremely scalable. It also enables horizontal scaling, which allows data to be dispersed across numerous database instances. This aids in data load balancing as well as data duplication for backup purposes.

Flexibility - MongoDB is schema-less, which means it doesn't enforce field relationships and instead allows the storage of any data type values in a stream. Because data is stored in a document/collection kind of model, it allows for on-the-fly data modelling, data change, and updating without the headache of schema updates that an RDBMS requires.



MongoDB - Features

Sharding - Sharding in MongoDB involves horizontally splitting data across multiple servers. This distributes workload, improves scalability, and allows for handling large datasets efficiently by spreading read and write operations across shards.

Data Replication and Recovery - In the event of a system breakdown, MongoDB provides specific tools for data replication as a backup. A main server keeps all of the data and handles read/write and other transactions. The secondary server is a replica of the primary server that can act as the primary server in the event that the first server crashes or fails, ensuring continuous high availability.

High Performance and Speed - MongoDB has a variety of capabilities that aid in faster processing, including dynamic ad-hoc querying, indexing for faster search functionality, and aggregating queries. In comparison to an RDBMS, querying is also faster because relevant data is kept together in documents. Even in high-volume, crucial circumstances, this ensures great performance.



MongoDB - Installation

MongoDB can be installed very easily on the 3 most used platforms: Ubuntu, MacOS and Windows.

Please choose the appropriate link for your OS and follow the steps :

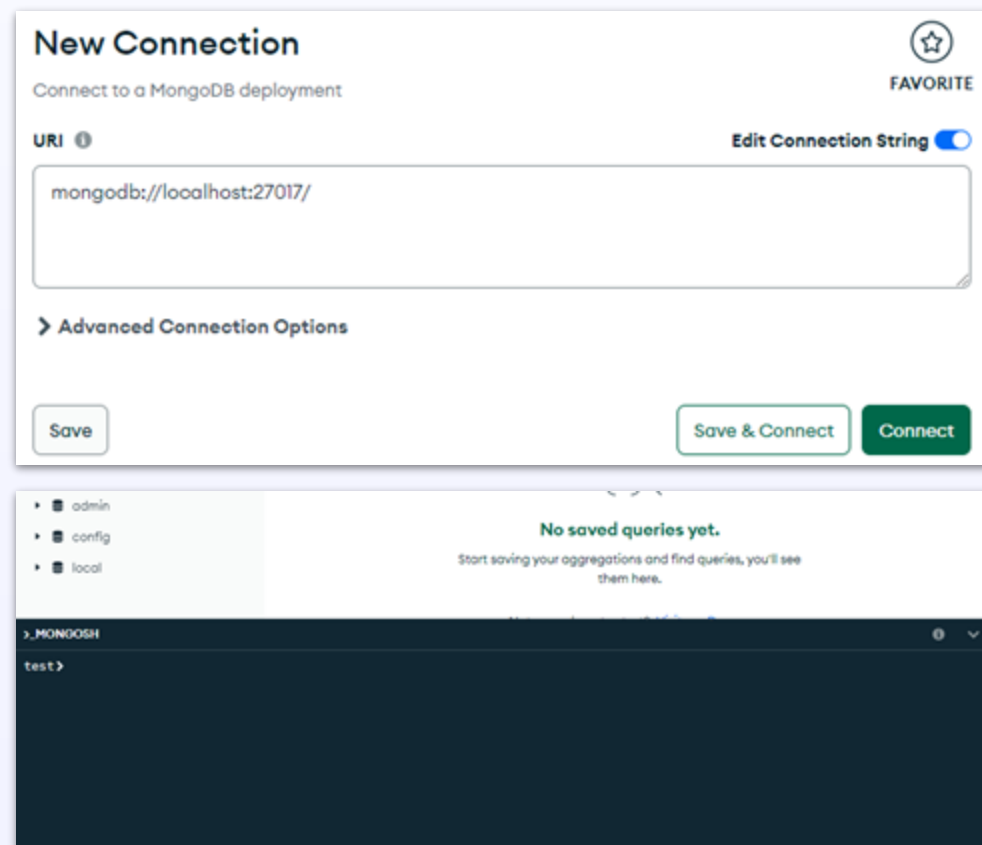
- ❖ [Ubuntu](#)
- ❖ [MacOS](#) (using Homebrew)
- ❖ [Windows](#)

Be sure to install the latest **Community Edition** of MongoDB. The default options should work in most cases.

Also install **MongoDB Compass**, which is a handy user interface tool we will use to visualise your MongoDB databases and run commands.



MongoDB - Connection



After installation is complete, the easiest way to interact with MongoDB is via **MongoDB Compass**.

Open Compass and click **Connect** (as shown left) to open a connection to your MongoDB server
(this needs to be installed and running first).

After connecting, expand the **MongoSH** window at the bottom (as shown left). This acts as a terminal where you can type and run commands on your Mongo databases.

MongoDB - Basic Access

You can see on the left of the UI that by default Mongo creates three databases: admin, config and local. These databases are for internal use so **don't** touch or delete them. We can also **list** the current databases from the terminal by running the following command via MongoSH:

```
show databases
```


```
>_MONGOSH
> show databases
< admin   40.00 KiB
   config  72.00 KiB
   local   80.00 KiB
test> |
```


Let's **create** our own database to add to this list. The following command creates a database called **newDB**:

```
use newDB
```

```
> use newDB
< switched to db newDB
newDB>
```


MongoDB - Basic Access

If you now refresh the list of databases in Compass (click the Refresh icon  +) you will see the **newDB** database added to the list, as it's still empty.

 Databases

 +

Since MongoDB as our DBMS supports multiple databases, we always need to specify **which** database we are currently working with. The `use` command we used for **creating** a database will also **switch** between existing databases, to let us change where our commands will be executed.

```
db
```

We can also check to see which is the currently selected database by simply running:

```
> db
< newDB
newDB> |
```

MongoDB - **CRUD** operations

What is CRUD ? Crud is a succinct way of describing the four essential storage operations:

Create, Read, Update, and Delete.

You should be familiar with these core operations if you want to work with databases. They apply in both SQL and NoSQL environments, although the syntax can vary greatly. We will look at each in turn.

Create Operations

MongoDB provides the following methods to insert documents into a collection:

❖ `db.collection.insertOne()`

❖ `db.collection.insertMany()`

In MongoDB, **collections** are equivalent to **tables** in the SQL world. They are **document** collections. Documents are essentially **objects**, similar to rows in a SQL database.

MongoDB - **CRUD** operations

The following command will **insert** (create) a new **document** (as represented by the **object** passed to the `insertOne` function), into the named collection `newCollection`:

```
db.newCollection.insertOne({name: "New Doc"})
```

The collection name needs to go after the `db.` and before the call to `insertOne`. If it doesn't yet exist, it will instantly be created as a new collection, just like we did with **newDB** above. So now we have our first collection: **newCollection**, thanks to this command.

You can only insert one document at a time with the `insertOne()` method. The document we inserted in this example is a very basic JSON object containing only one field, called 'name'.

```
> db.newCollection.insertOne({name: "New Doc"})
< {
  acknowledged: true,
  insertedId: ObjectId("65b74dc43b1ce0ef368eb089")
}
newDB>
```

MongoDB - **C**RUD operations

With same logic, we can insert more than one document at a time with the `insertMany()` method, into the relevant collection. Whereas `insertOne` takes a **single** object as a parameter, `insertMany` takes an **array** of objects, all of which will be inserted as new documents in the specified collection:

```
db.newCollection.insertMany([{name: "New Doc #2"}, {name: "New Doc #3"}])
```

Now we can refresh our list of databases, and see our **newDB** in the list. Expand the arrow next to it to show the list of collections, and click on our **newCollection** to show all of the inserted documents.

Try inserting a new document representing yourself, with name, age and location fields!

You can also see all the collections by using the command:

```
show collections
```

```
> show collections
< newCollection
newDB >
```

MongoDB - **CRUD** operations

Read Operations

Documents can be retrieved from a collection using **read** operations. In MongoDB, we use the **find** method to do this ([see here for more](#)):

❖ `db.collection.find()`

We can run it without any parameters, in which case it will match all documents in the collection and return all their fields. It also takes three optional object parameters:

- **query** - filters documents based on a query condition to match only certain records
- **projection** - controls which fields of each document will be returned (default all)
- **options** - specifies options to modify query behavior and how results are returned

❖ `db.collection.find(query, projection, options)`

MongoDB - CRUD operations

Find Operations

```
db.newCollection.find()
```

```
> db.newCollection.find()
< {
  _id: ObjectId("65b74dc43b1ce0ef368eb089"),
  name: 'New Doc'
}
{
  _id: ObjectId("65b750683b1ce0ef368eb08a"),
  name: 'New Doc #2'
}
{
  _id: ObjectId("65b750683b1ce0ef368eb08b"),
  name: 'New Doc #3'
}
```

There is another field called `_id` as well as the name that we inserted. MongoDB creates a unique `_id` for every document, which acts as a primary key and can be used for manipulating that document. Used like this, `find` returns every document in our `newCollection`. *Try it!*

MongoDB - CRUD operations

Find Operations with Query

We can also use the **query** parameter to look for a document that has a name equal to “New Doc”:

```
db.newCollection.find({name: "New Doc"})
```

```
> db.newCollection.find({name: "New Doc"})  
< {  
  _id: ObjectId("65b74dc43b1ce0ef368eb089"),  
  name: 'New Doc'  
}
```

The **query** parameter also supports various operations besides straight equality, such as greater than, less than, not equal, etc.

For more information check the [MongoDB documentation here](#).

This command will find all documents with a name not equal to ‘New Doc’. *Try it!*

```
db.newCollection.find({ name: { $ne: "New Doc" } })
```

MongoDB - CRUD operations

Find Operations with Query and Projection

The second **projection** parameter allows us to specify certain fields to be returned, by setting them to **zero** (not returned) or **one** (is returned):

```
db.newCollection.find({name: "New Doc"}, {_id: 0})
```

```
> db.newCollection.find({name: "New Doc"}, {_id: 0})  
< {  
  name: 'New Doc'  
}
```

The document returned from the above query will contain all fields **except** the `_id`. Changing the 0 to 1 will return **only** the `_id` field, for each document matched by the query parameter (first). *Try it!*

To include both `_id` and `name` fields, simply set both to 1 in the **projection** parameter: *Try it!*

```
db.newCollection.find({name: "New Doc"}, {_id: 1, name: 1})
```


MongoDB - CRUD operations

Update Operations

You can update the documents in a collection with following commands. **updateOne** will only update a single document (the first if multiple match) while **updateMany** will update all matching documents.

- ❖ `db.collection.updateOne()`

- ❖ `db.collection.updateMany()`

Both methods take at least two parameters: filter is an object used to specify **which documents** should be updated, and update is an object containing **what changes** the update consists of.

- ❖ `db.collection.updateOne(filter, update, options)`

options is an object of optional configuration for the update. You can see all of the options [here](#). The **upsert** option is probably the most useful, as it handles both inserting and updating in one operation, depending on whether there is a record matching the **filter**.

MongoDB - CRUD operations

Update One Operation

The following command will **update** a single document with a name of 'New Doc' and set a new field **number** to a value of 5:

Try it!

```
db.newCollection.updateOne({name: "New Doc"}, {$set: {number: 5}})
```

First parameter is the **filter** parameter, specifying that we are looking for documents with a **name** of "New Doc".

In the second parameter we use the special Mongo operator **\$set** to specify what needs to be updated. If the key we are trying to update (**number**) exists, it will be updated to the new value (5) and if it doesn't exist, it will be created. *Try using **find** to check the result of your update!*

```
> db.newCollection.updateOne({name: "New Doc"}, {$set: {number: 5}})
< {
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

MongoDB - CRUD operations

Update Many Operation

The following command will **update** potentially many documents with a name not equal to 'New Doc' and set a new field **description** to a value of 'updated': *Try it!*

```
db.newCollection.updateMany({name: {$ne: "New Doc"}}, {$set: {description: "updated"}})
```

First parameter is the **filter** parameter, specifying that we are looking for documents that do not have a **name** of "New Doc".

In the second parameter we use the special Mongo operator **\$set** to specify that the description field will be set to 'updated' in all matching documents. *Try using **find** to check the result of your update!*

```
> db.newCollection.updateMany({name: {$ne: "New Doc"}}, {$set: {description: "updated"}})
< {
  acknowledged: true,
  insertedId: null,
  matchedCount: 2,
  modifiedCount: 2,
  upsertedCount: 0
}
```

MongoDB - CRUD operations

Delete Operations

You can remove the documents in a collection with following commands. **deleteOne** will only remove a single document (the first if multiple match) while **deleteMany** will remove all matching documents.

- ❖ `db.collection.deleteOne()`

- ❖ `db.collection.deleteMany()`

Both methods take at least one parameter: filter is an object used to specify **which documents** should be removed, and options is an object of optional configuration for the removal.

- ❖ `db.collection.deleteOne(filter, options)`

Use an empty object {} as the first filter parameter to delete either the **first** or **all** documents in the collection without filtering them.

MongoDB - CRUD operations

Delete One Operation

```
db.newCollection.deleteOne({name: "New Doc"})
```

```
> db.newCollection.deleteOne({name: "New Doc"})  
< {  
  acknowledged: true,  
  deletedCount: 1  
}
```

If there is more than one document that matches the query, only the first will be deleted using the `deleteOne` method.

As you might expect, `deleteMany()` will eliminate all documents that meet the query. *Try it!*

MongoDB - CRUD operations

Drop Collection

The following command will delete our **newCollection** entirely, along with any documents inside it:

```
db.newCollection.drop()
```

Drop Database

Finally, there is a fast command that we can use to delete the database we just established. Make sure you're working in the appropriate database before you delete. To check the current database, type `db`, and to switch, type `use newDB`.

Drop Database Operation

```
db.dropDatabase()
```

```
> db.dropDatabase()  
< { ok: 1, dropped: 'newDB' }
```

After running **dropDatabase**, we no longer have our **newDB** database.

Exercise 2

Create your own MongoDB database for a general Blogging application and test it with documents in each collection. You should use the logical model from **Exercise 1** as a reference to complete this task.

Requirements:

- ❖ The system should store users (using basic user-related fields)
- ❖ The users should be able to create multiple posts (posts should be very basic with title, description and image)
- ❖ Other users should be able to like the posts and comment on the posts.

Section 4:

Introduction to MySQL



MySQL is a database management system for relational databases (RDBMS), which means it creates a database with tables, columns, rows, and indexes. It is an open-source database with a large community and worldwide support.

MySQL is a database engine that implements Structured Query Language (SQL). SQL is an established query language that is used to manipulate and extract data in relational databases.

MySQL - Structured Query Language (SQL)

A relational database's basic language is SQL, which is used to access and manage the database. SQL allows you to add, edit, and delete rows of data, get subsets of data, modify databases, and conduct a variety of other tasks. The different subsets of SQL are as follows:

- ❖ **DDL** (Data Definition Language) - allows you to do things like CREATE, ALTER, and DELETE items in the database such as tables and indexes.
- ❖ **DML** (Data Manipulation Language) — allows you to manipulate and access data. It aids in the inserting, updating, deleting, and retrieval of data from a database (CRUD).
- ❖ **DCL** (Data Control Language) — allows you to control database access, by granting or revoking user access permissions, for example.
- ❖ **TCL** (Transaction Control Language) — allows you to work with individual database transactions. Commit, Rollback, Savepoint, and Set Transaction are some examples.

MySQL - Features

Some of the features of MySQL include:

- ❖ **Ease of Management** – The software is simple to install and employs an event planner to automatically schedule jobs.
- ❖ **Robust Transactional Support** — Maintains the ACID (Atomicity, Consistency, Isolation, and Durability) properties of any RDBMS while also supporting distributed multi-version support.
- ❖ **Integrated Application Development** — MySQL comes with plugin libraries that allow you to integrate the database into any application. For application development, it also provides stored procedures, triggers, functions, views, and many more features.
- ❖ **High Performance** — With different memory caches and table index partitioning, it provides quick load utility.
- ❖ **Low Total Cost Of Ownership (TCO)** - This lowers licence and hardware costs.

MySQL - Features

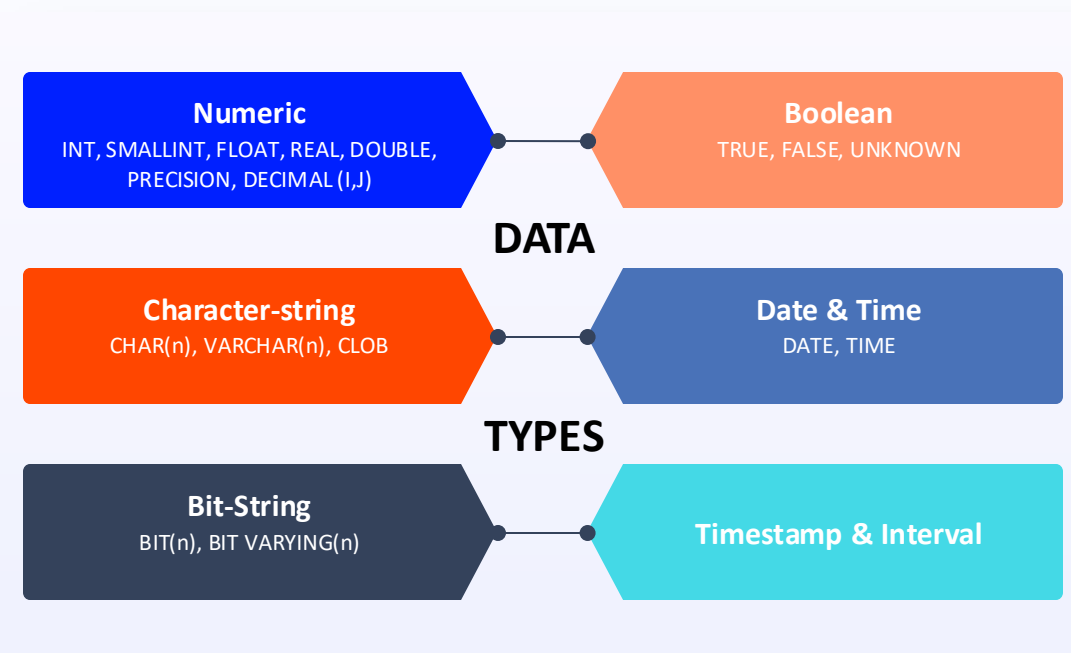
- ❖ **Open Source & Round-the-Clock Assistance** — This RDBMS is open source and offers round-the-clock support for both open source and enterprise editions.
- ❖ **Secure Data Protection** - MySQL has a number of sophisticated procedures in place to ensure that only authorised users have access to databases.
- ❖ **High Availability** — MySQL supports high-performance master/slave replication as well as cluster servers.
- ❖ **Scalability and Flexibility** — MySQL allows you to run deeply integrated applications as well as develop massive data warehouses.



MySQL - Data Types

An important part of creating a physical model is choosing appropriate data types for each of the entity attributes. Let's see some of the data types supported by MySQL ([see here for more detail](#)):

- ❖ **Numeric** — **INT**egers of various sizes, **FLOAT**ing-point (real) numbers of varying precisions, and formatted numbers are all supported and included in this data type.
- ❖ **Character-string** — These data types have a set amount of characters (**CHAR**) or a variable number of characters (**VARCHAR**). Character Large Object (**CLOB**) is a variable-length string that is used to describe columns with large text values, often read from text-based files.



MySQL - Data Types

- ❖ **Boolean** — This data type has two possible values: TRUE or FALSE. Because SQL has NULL values, UNKNOWN is used as a three-valued logic.
- ❖ **Date & Time** — **DATE** stores a date with no time, using the format YYYY-MM-DD for YEAR, MONTH, and DAY. Similarly, **TIME** data types store HOUR, MINUTE, and SECOND components in the format HH:MM:SS. **DATETIME** combines both of the above to store both date and time.
- ❖ **Timestamp & Interval** — In addition to the DATE and TIME fields, the **TIMESTAMP** data type provides a minimum of six places for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. A relative value can be used to increment or decrement an absolute value of a date, time, or timestamp, as shown by the **INTERVAL** data type.
- ❖ **Bit-string** — These data types can be either fixed or variable in length. Binary Large Object (**BLOB**) is a variable-length bit-string data type that can be used to express columns that have large binary values, such as images. Generally though, it is more efficient to store images as string URLs.

MySQL - Installation

MySQL can be installed very easily on the 3 most used platforms: **Ubuntu**, **MacOS** and **Windows**.

Please choose the appropriate link for your OS and follow the steps to install the latest Community Edition :

- ❖ [Ubuntu](#)
- ❖ [MacOS](#) (DMG installer, [HomeBrew](#) is another option)
- ❖ [Windows](#)

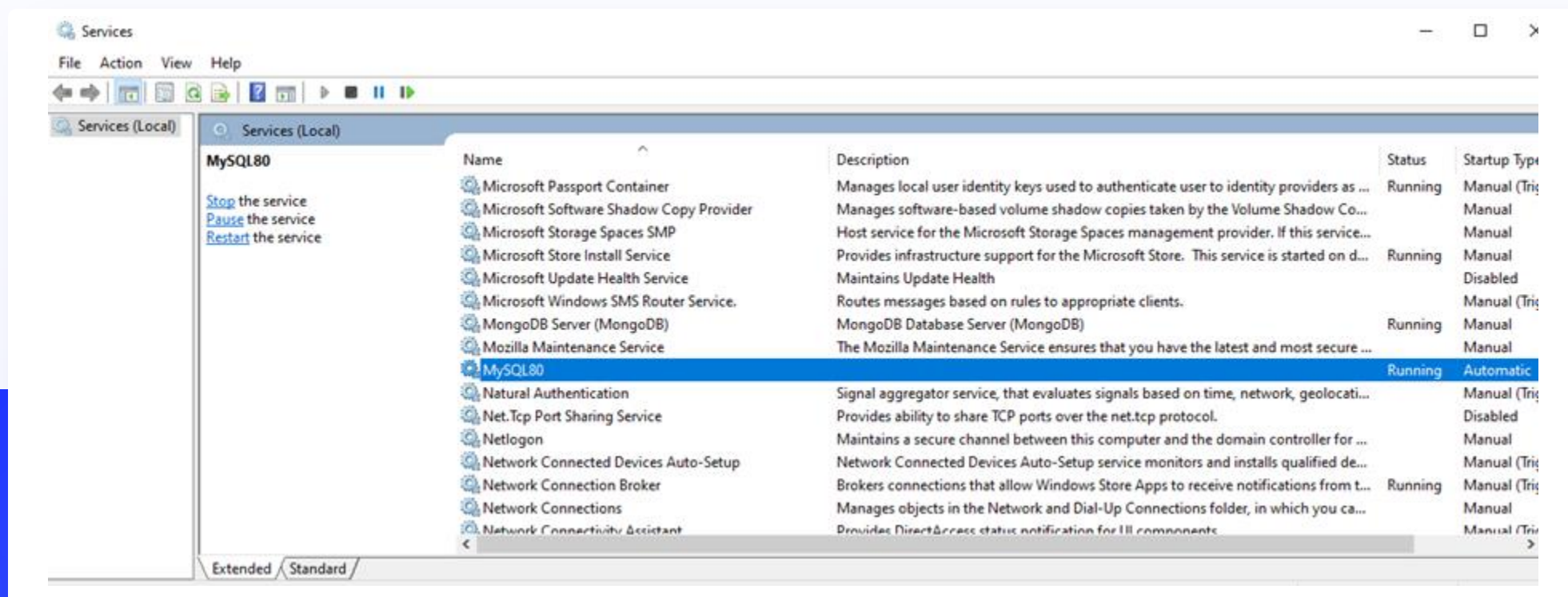
Installation tips:

- ❖ If it asks you to create an Oracle account, **ignore it** and click 'No thanks, just start my download'.
- ❖ Choose a **Custom** setup from within the installer. The only applications you need to install are **MySQL Server** and **MySQL Workbench** (a handy user interface tool we will use to visualise your MySQL databases and run commands).
- ❖ When asked to set a root password, ***make sure you record it somewhere***, as you will need to know it later and it is difficult to reset. Otherwise the defaults should be fine.

MySQL - Basic Access

After installation is complete, we first need to ensure that MySQL Server is running.

On **Windows**, search for the **Services** app and open it to display a list of all installed apps. Scroll down to find MySQL80 (or similar) and make sure its **Status** is **Running**:



MySQL - Basic Access

On **MacOS**, if you have used the DMG file to install, follow these instructions to ensure that the MySQL Server is running and will start automatically when your computer starts:

[MacOS MySQL Launch Instructions](#)

If you used HomeBrew to install, just run this command to start MySQL Server:

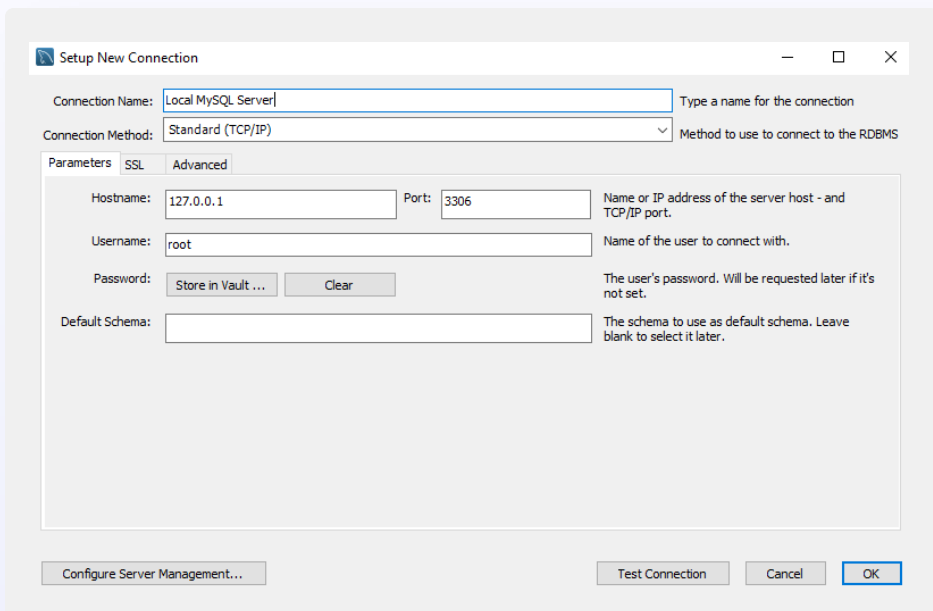
```
brew services start mysql
```

On **Ubuntu**, your MySQL Server should automatically start after installation. If you have issues please refer to the [Linux installation guides](#).

MySQL - Basic Access

Once MySQL Server is running, we will use **MySQL Workbench** to connect to it and view our databases. If you have not installed this program already, please [download it from here](#), choosing the appropriate operating system.

Open Workbench and set up a new connection to your local server:



Default parameters should be right:

Hostname - 127.0.0.1

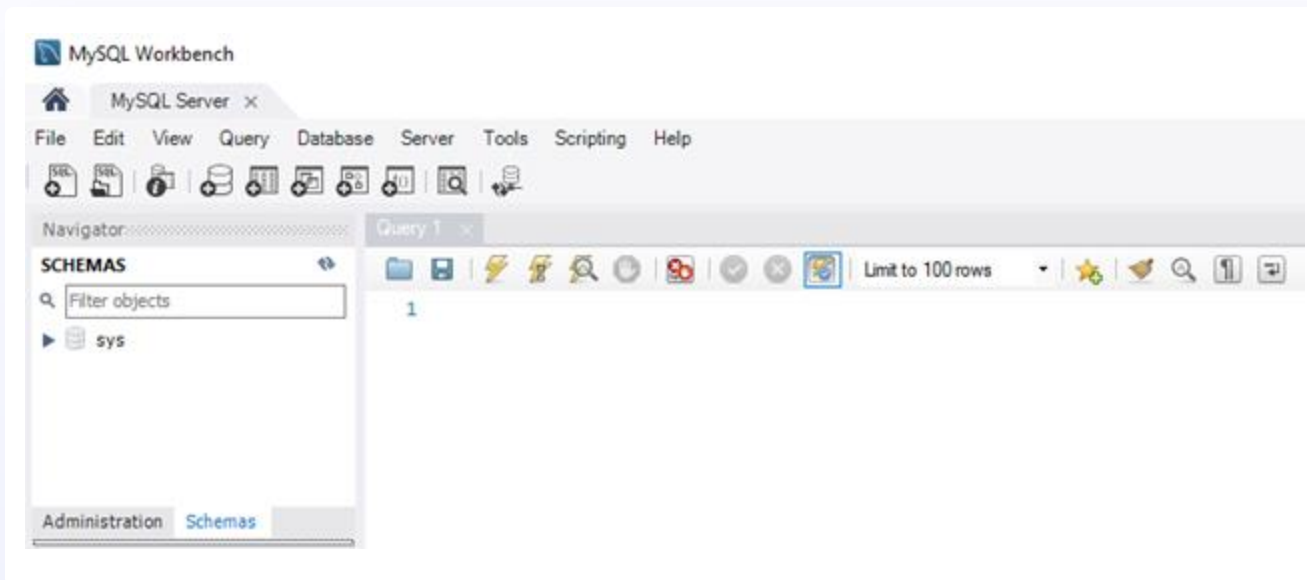
Port - 3306

Username - root

Click **OK** - this will prompt you for the root password you saved from the installation process. Store it in your vault.

MySQL - Basic Access

A MySQL server can host several databases, each with its own set of tables. By default, MySQL creates four databases: `information_schema`, `mysql`, `performance_schema` and `sys`. These databases are for MySQL's internal use, so please don't touch them or delete them.



Use the **Schemas** tab on the left to see a list of all MySQL Databases.

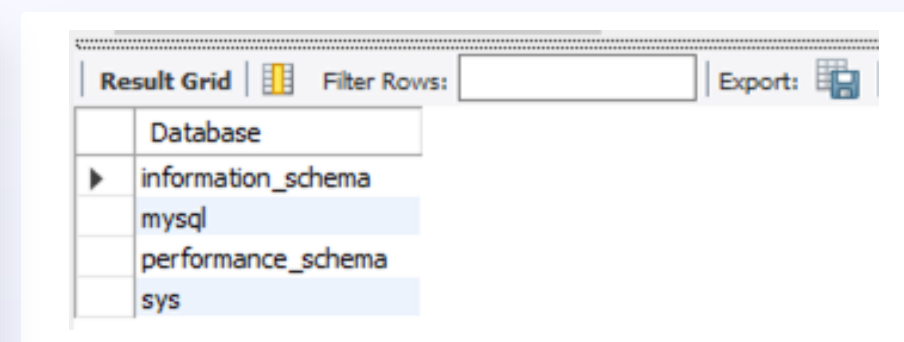
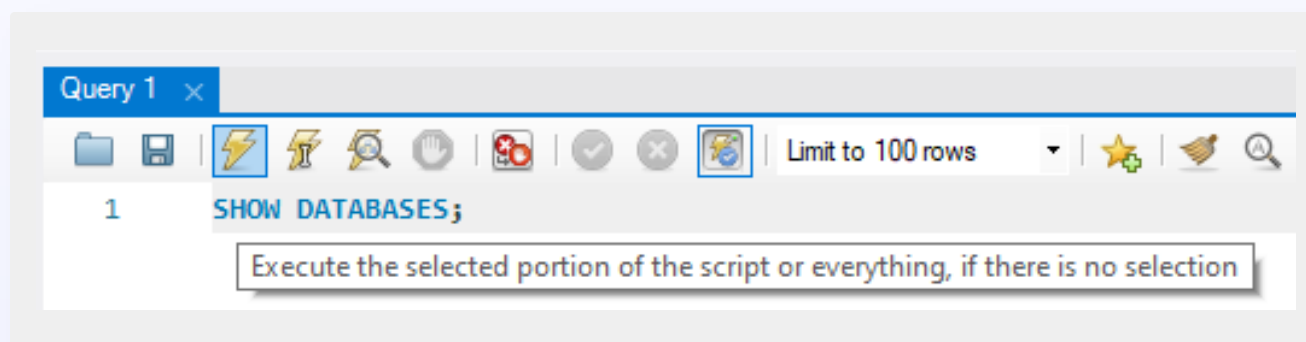
The **Query** window in the center is where we can write SQL commands to execute on our databases.

MySQL - Basic Access

Let's run our first SQL command, to simply print a list of all databases within our MySQL DBMS. By convention, all SQL keywords are written in **UPPERCASE**, with custom database, table and column names in **lowercase**. Individual SQL commands should end with a semi-colon.

```
SHOW DATABASES;
```

Type the above command into the **Query** window in Workbench, then click the **lightning bolt icon** to execute it. This should show the **Result Grid** in the second screenshot below, listing all the databases:



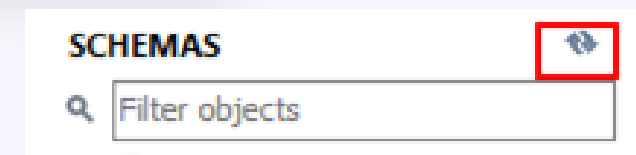
MySQL - Basic Access

```
CREATE DATABASE newDB;
```

After running this command in the **Query** window in Workbench, you should see the following successful result in the **Output** window underneath:

	17 11:25:53 CREATE DATABASE newDB	1 row(s) affected
---	-----------------------------------	-------------------

Clicking the refresh icon in the left-hand Schemas tab should now show the new database in the list of schemas:



The next step is to **select** our new database as our working, or active, database. To do this, either double-click the 'newDB' database name in the left-hand list of schemas, or run this SQL:

```
USE newDB;
```

MySQL - Basic Access

Now let's create a **table** in our current database. We can do this by using the CREATE TABLE SQL command, and specifying the desired **columns** and **constraints** with the right syntax.

Let's create a table named 'movies' with four columns — 'id', 'title', 'director', 'genre' as follows, to match a physical data model that may look like this: >

```
CREATE TABLE movies (id INT NOT NULL, title VARCHAR(50) NOT NULL, director VARCHAR(50), genre VARCHAR(50), PRIMARY KEY(id));
```

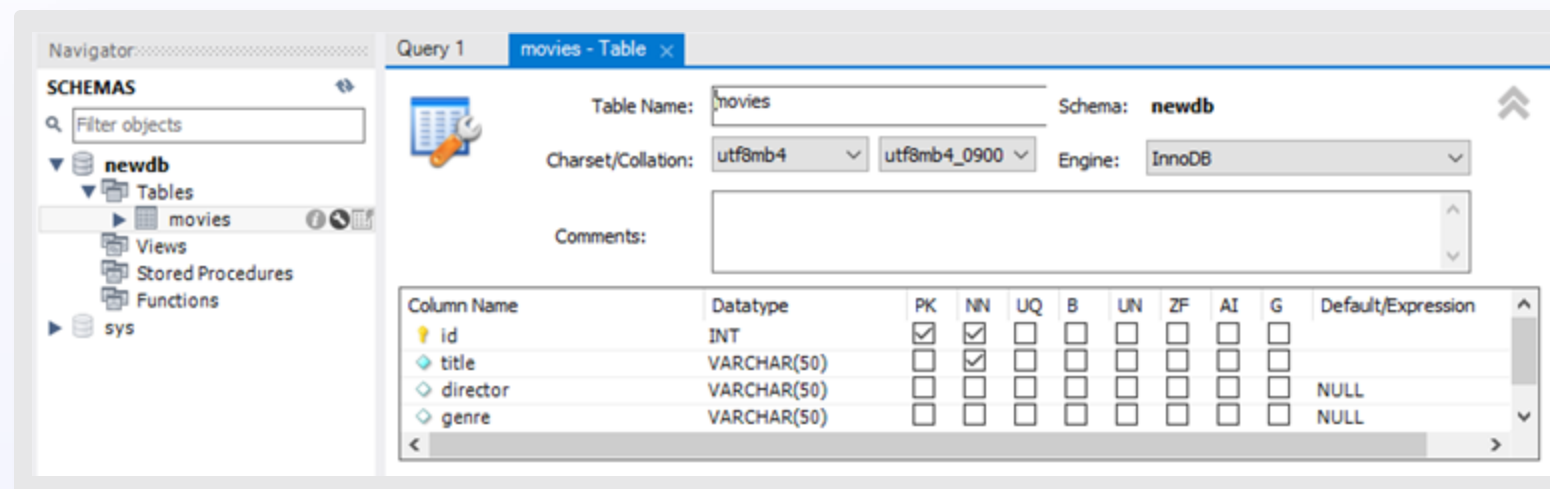
movies	
PK	<u>id</u> : INT
	title : VARCHAR
	director : VARCHAR
	genre : VARCHAR

Refresh the schemas again to see the new table in our **newDB** database, or run `SHOW TABLES`. You can also run `DESC movies` to see the description of the new table.

By setting up a schema like this, we are defining **constraints** (or rules) for the data inside this table that specify each movie (row in the table) needs to have a **unique, numeric id** and a **text title**, and may optionally have a **text-based director** and a **genre**, none of which exceed **50 characters** in length.

MySQL - Basic Access

We can also use Workbench to analyse and update the schema of each table. Find and click the 'movies' table in the Schemas tab on the left. When hovering over the table, you will see **3 extra icons** appear. The first is for **table info**, with various tabs that provide detail about the table. The second is the spanner for **table config**, which lets us modify the schema:



From here we can easily add new columns and modify constraints. A handy constraint for primary keys is the AUTO INCREMENT feature, enabled by ticking the **AI** box for the **id** column.

MySQL - **CRUD** operations

Now that we have a table, let's add data by having a look at some of the CRUD operations with MySQL.

Create Operations ([see here for more tips](#))

In MySQL, we use the `INSERT INTO` command to add a new record to a table, specifying the table name and individual column values. If all of the values are in the same order as the table's columns, we don't need to specify the column names.

```
INSERT INTO movies VALUES (1, "Troy", "Wolfgang Peterson", "Historical Fiction");
```

We can also add multiple rows at the same time, by adding several sets of values separated by commas:

```
INSERT INTO movies VALUES (2, "Inception", "Christopher Nolan", "Science Fiction"), (3, "The Greatest Showman", "Michael Gracey", "Musical Drama");
```

If the **id** field is set to **auto increment**, and the director and genre fields are **not required**, we can also insert a new movie by simply specifying its **title**:

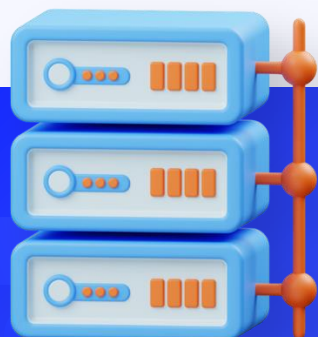
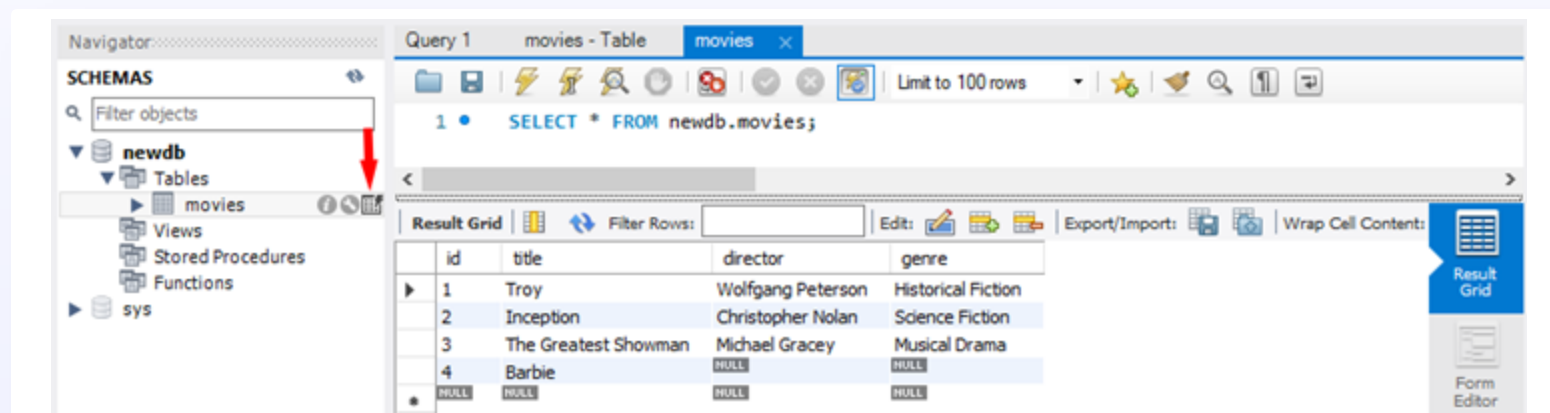
```
INSERT INTO movies (title) VALUES ("Barbie")
```

MySQL - CRUD operations

The Output window in Workbench should provide detail on the outcome of the INSERT statements on the previous slide, eg:

✓	22	12:40:39	INSERT INTO movies VALUES (1, "Troy", "Wolfgang Peterson", "Histori...	1 row(s) affected
✓	23	12:40:47	INSERT INTO movies VALUES (2, "Inception", "Christopher Nolan", "Sc...	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0
✗	24	12:42:50	INSERT INTO movies (title) VALUES ('Barbie')	Error Code: 1364. Field 'id' doesn't have a default value
✓	25	12:42:58	Apply changes to movies	Changes applied
✓	26	12:43:05	INSERT INTO movies (title) VALUES ('Barbie')	1 row(s) affected

To easily examine all of the data in a table, we can also use the third of the icons when hovering over a table in the Schemas tab, to list **table data**:

Query 1 movies - Table movies

1 • SELECT * FROM newdb.movies;

id	title	director	genre
1	Troy	Wolfgang Peterson	Historical Fiction
2	Inception	Christopher Nolan	Science Fiction
3	The Greatest Showman	Michael Gracey	Musical Drama
4	Barbie	NULL	NULL
5	Barbie	NULL	NULL

MySQL - **CRUD** operations

Read Operations ([see here for more tips](#))

The 'SELECT' clause is used for **retrieving** records from a table. There are two main ways to use it:

- ❖ `SELECT column1, column2, column3... FROM table_name;` (specifying which columns)
- ❖ `SELECT * FROM table_name;` (using wildcard to select data from all columns)

The data will be selected from the table mentioned after the 'FROM' clause. *Try using both types of syntax on our movies table, to select all columns, and to select only the title and genre!*

The 'WHERE' clause is used in conjunction with the SELECT clause to **filter** the records and select only those that match a given condition, eg:

```
SELECT * FROM movies WHERE title LIKE 'T%';
```

The **WHERE** clause supports several standard **operators** when comparing data: `=` (equal), `>/<` (greater than/less than), `>=`/`<=` (greater than or equal/less than or equal), `<>` (not equal), **BETWEEN**, **LIKE**, **IN**.

Refer [here](#) and try to select all records with an id greater than (or equal to) 2!

MySQL - CRUD operations

Read Operations cont.

The SELECT clause is an extremely powerful one and we won't explore all possible features. The following are worth mentioning and researching further, especially if pursuing back-end development:

- ❖ The **DISTINCT** keyword, when added after SELECT, ensures that no duplicate values are selected. It is useful for getting a list of all unique categories, countries, or genres, from a table with multiple records.

```
SELECT DISTINCT genre FROM movies;
```

- ❖ When using the LIKE operator in a WHERE clause, the % character can be used as a **wildcard** to do partial matching, such as the example on the previous slide to find records with titles starting with T.

```
SELECT title, director FROM movies WHERE genre LIKE '%Drama';
```

- ❖ MySQL includes many different [functions](#) useful for aggregating and operating on the SELECTed records:

```
SELECT COUNT(id) AS num_movies FROM movies;
```

- ❖ The AS keyword can be used as above to set or modify the name of a returned column by providing an alias.

MySQL - CRUD operations

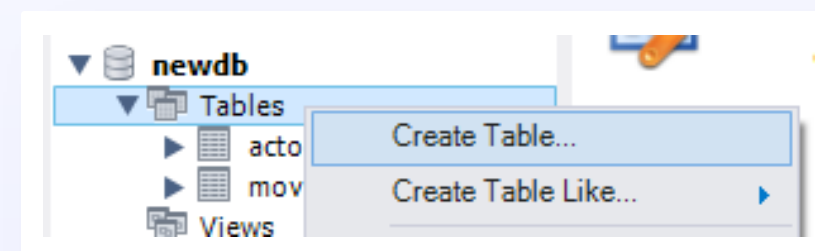
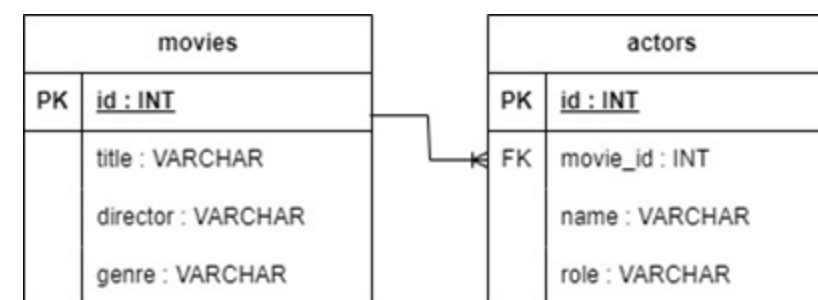
Another important operation commonly done in SQL is to **combine data** from two different tables together in the same query. This is done by utilising the relationships set up using **foreign keys**.

First we will create a second 'actors' table matching this physical database model, representing a simple **one-to-many** relationship between movies and actors, so that one movie has multiple actors:

We can create the table and the foreign key constraint using SQL:

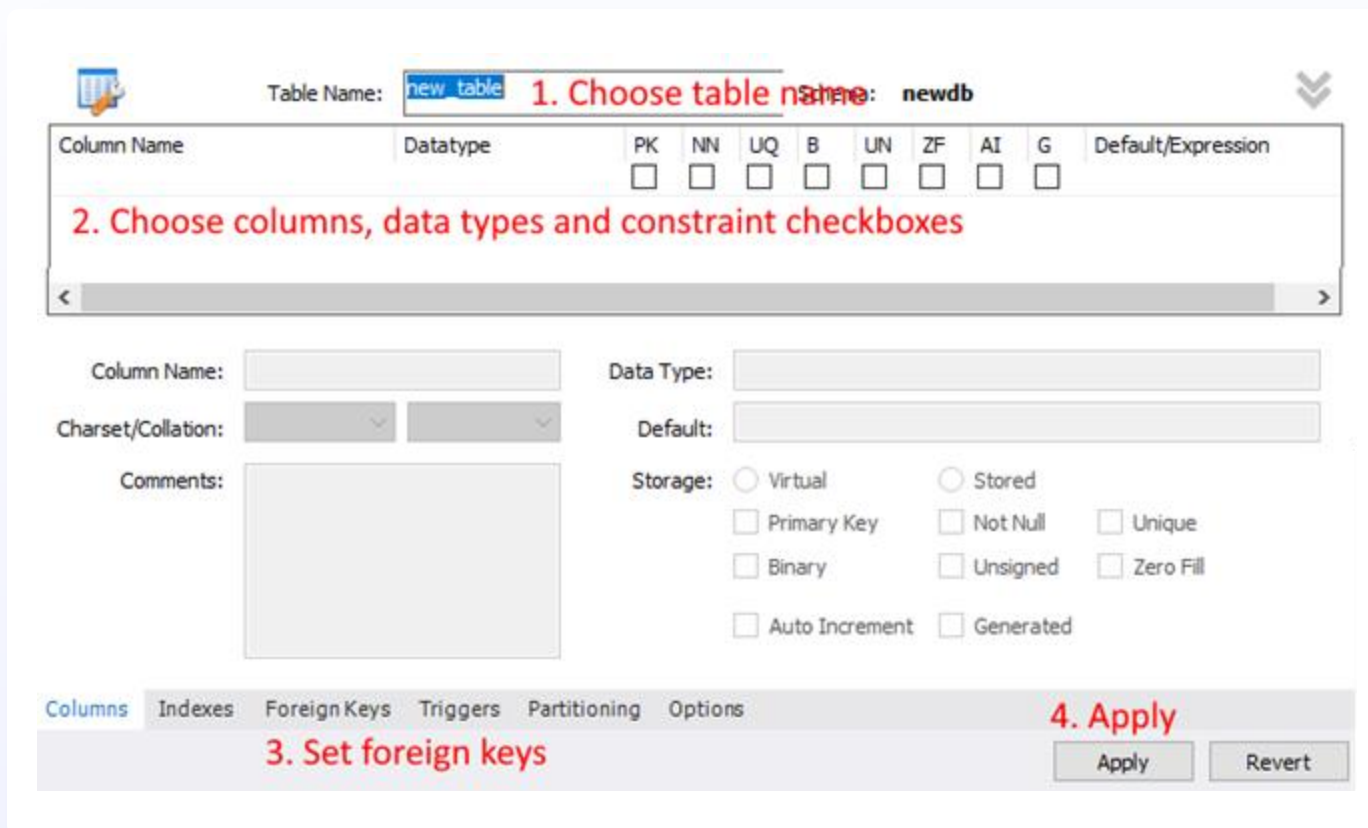
```
CREATE TABLE actors (id INT NOT NULL AUTO_INCREMENT, movie_id INT NOT NULL,
name VARCHAR(100) NOT NULL, role VARCHAR(100) NOT NULL, PRIMARY KEY(id), FOREIGN KEY (movie_id) REFERENCES
movies(id));
```

We can also use Workbench to create the table and the foreign key. Right-click on **Tables** in the Schemas tab and choose **Create Table...** from the context menu.



MySQL - CRUD operations

From here we can visually do everything that the `CREATE TABLE` SQL command does:



The image shows the MySQL Table Designer interface with the following elements and annotations:

- Table Name:** `new_table` (Annotated with "1. Choose table name")
- Schema:** `newdb`
- Column List:** A table with headers: Column Name, Datatype, PK, NN, UQ, B, UN, ZF, AI, G, Default/Expression. (Annotated with "2. Choose columns, data types and constraint checkboxes")
- Column Configuration:**
 - Column Name:
 - Data Type:
 - Charset/Collation:
 - Default:
 - Comments:
 - Storage: ☐ Virtual ☐ Stored
 - ☐ Primary Key ☐ Not Null ☐ Unique
 - ☐ Binary ☐ Unsigned ☐ Zero Fill
 - ☐ Auto Increment ☐ Generated
- Tabs:** Columns, Indexes, Foreign Keys, Triggers, Partitioning, Options. (Annotated with "3. Set foreign keys")
- Buttons:** Apply, Revert. (Annotated with "4. Apply")

First type the table name in the top field.

Next click under **Column Name** to add a new column and select the appropriate data type and constraint checkboxes.

Finally click the Foreign Keys tab at the bottom to set up foreign key relationships.

When complete, click the Apply button.

MySQL - **CRUD** operations

Now that we have two linked tables, we can see how to **JOIN** them together to select various data combinations, such as all the actors for a given director, or all the movies with a given actor.

First let's insert some actors (*try inserting some more of your own!*):

```
INSERT INTO actors VALUES (1, 1, 'Brad Pitt', 'Achilles'), (2, 2, 'Leonardo DiCaprio', 'Cobb');
```

There are many potentially complex ways to join our two tables ([see here for more examples](#)) but the most simple can be done using the JOIN clause as follows:

```
SELECT * FROM movies JOIN actors ON movies.id = actors.movie_id;
```

This selects all columns from both tables and maintains the link so that the right actors are listed for the right movies. We can replace the * wildcard with specific column names to return only the specified columns, and we can add a WHERE clause at the end to filter our records with a given condition. *Try it!*

MySQL - CRUD operations

Update Operations ([see here for more tips](#))

When we need to make changes to existing records, we use the UPDATE clause. A typical update command would follow the format:

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

```
UPDATE movies SET genre = 'War Film' WHERE id = 1;
```

We can update the **genre** of the **movie** with id **1** using the following example:

An UPDATE command without a WHERE clause will update all records in the table, so it is usually important to include one so that only relevant records are updated.

We can update more than one column with the same UPDATE command by listing all columns and their new values, separated by commas, after the SET clause, eg:

```
UPDATE movies SET genre = 'Drama', director = 'Fake Director' WHERE id >= 3
```

MySQL - CRUD operations

Update Operations cont.

MySQL also includes a whole range of built-in **functions** that are useful when updating, to handle things like combining column and string values together (CONCAT), partly replacing column values with updated values (REPLACE), or identifying and using partial strings (SUBSTRING).

```
UPDATE movies SET title = CONCAT(title, ' 2') WHERE id > 2;
```

This command will update the titles for movies 3 and 4 by adding ' 2', turning them into sequels.

```
UPDATE movies SET genre = REPLACE(genre, 'Fiction', 'Story') WHERE id <= 2;
```

This command will change the genre by replacing the word 'Fiction' with 'Story' for movies 1 and 2.

There are MANY other functions available, with more [details available here](#). Some like the above are useful for **text**-based columns, and there are many more for **number** and **date** based fields. Some are similar to JavaScript functions but will usually have a different syntax. *Try some!*

MySQL - CRUD operations

Delete Operations ([see here for more tips](#))

The **DELETE** clause is used in conjunction with the WHERE clause to delete one or more records. The general syntax for a delete command would look like 'DELETE FROM table_name WHERE condition;'

```
DELETE FROM actors WHERE id = 2;
```

If we leave out the WHERE clause, the command will delete all records from the table, so take care if this is not the intended behaviour.

Another option for deleting all records in a table is the TRUNCATE command:

```
TRUNCATE actors;
```

To test if the commands have worked as expected, simply run a SELECT command to view all records in the table. You can also use the **table data** icon which is the third that appears when hovering over the table name in Workbench.

MySQL - Operations

Drop a Table or Database

You can also delete a table, or even the whole database, using the **DROP** command:

`'DROP TABLE <table name>' or 'DROP <database name>'`

```
DROP TABLE movies;
```

You will notice when running the above command that it gives an error: *'Error Code: 3730. Cannot drop table 'movies' referenced by a foreign key constraint...'*. Since there is a foreign key in the actors table that references the **movies** table, it won't let us delete the movies table while this reference exists. If we **first** delete the actors table (`DROP TABLE actors;`), then the movies table is no longer referenced, and can be successfully deleted.

```
DROP DATABASE newDB;
```

The above command will remove our testing database newDB completely.

Using the Refresh icon in the Schemas tab after each command should reflect the changes.

Exercise 3

Create your own MySQL database for a general Blogging application. You should use the physical and logical model from **Exercise 1** as a reference to complete this task.

Requirements:

- ❖ The system should store users (using basic user-related fields)
- ❖ The users should be able to create multiple posts (posts should be very basic with title, description and image)
- ❖ Other users should be able to like the posts and comment on the posts.

Section 5:

Introduction to Redis



Redis is an **in-memory data structure store** that may be used as a database, cache, and message broker. It is open-source (BSD licensed) and stands for **R**emote **D**ictionary **S**erver.

Strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geographical indexes with radius searches, and streams are all supported data structures.

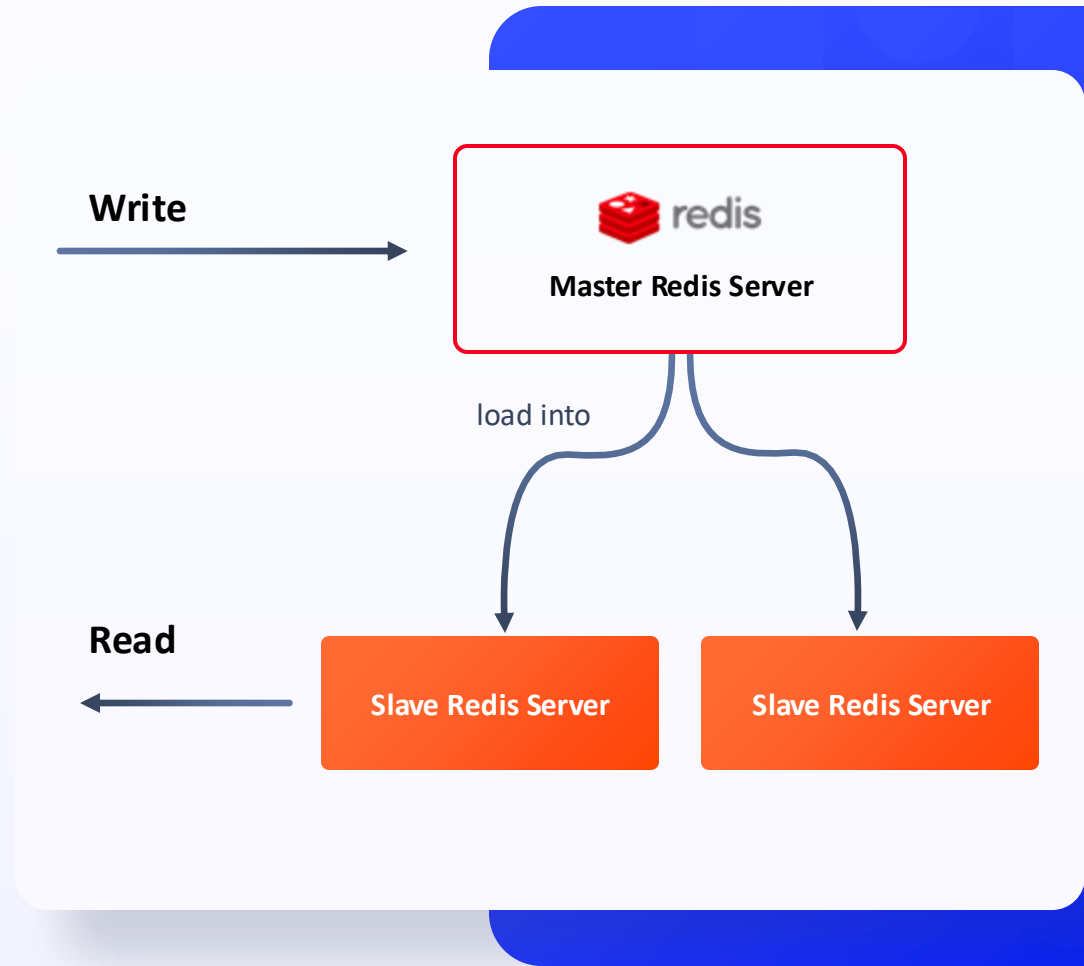
You can find the [official Redis information here](https://redis.io/).

Redis - Key-Value Store

Data is stored as a **key-value pair** in the Redis server/store. For instance, "Name"="John". The **key** is "Name" and the **value** is "John". Redis may be used as a NoSQL database because it can read and write data.

We can **WRITE** data to the Redis server using `SET name "John"` and also we can **READ** data from the server using `GET name`. There are other commands for writing and reading data in more complex ways as well.

It's a non-relational NoSQL database. This means that, unlike MySQL and Oracle databases, it lacks table, row, column, function, and procedure structures. It also doesn't employ SELECT, UPDATE, INSERT, or UPDATE statements.



Redis - Common Uses

Redis is used in a wide range of applications to **optimise performance** when accessing and analysing large amounts of data. Its speed is due to the way it primarily manages data **in-memory** instead of **on-disk**.

Some common applications for Redis include:

- ❖ **Caching:** Storing frequently accessed data in memory to improve application performance.
- ❖ **Session Management:** Storing user session data for web applications, ensuring fast access and scalability.
- ❖ **Message Queues:** Implementing lightweight message queues for task management and communication between components.
- ❖ **Real-time Analytics:** Collecting and analysing real-time data streams for monitoring and decision-making.
- ❖ **Pub/Sub Messaging:** Facilitating publish/subscribe messaging patterns for real-time communication between clients and servers.

Redis - Features

Some of the benefits and features of Redis include:

- ❖ **Simple and Flexible** — Because Redis is a NoSQL data store, we don't need to specify tables, rows, or columns. There's no need for schemas or constraints, and read/write operations are simple.
- ❖ **Durability** — Redis primarily operates on data in memory/cache. It does, however, have the ability to write to the disc, meaning we can utilise Redis as a full-fledged database or as a caching system.
- ❖ **Compatibility** — One of the most common uses of Redis is as a supplementary database for your apps to speed up transactions. A cache, like Redis, provides a high-speed data store, whereas a database is slower but more dependable and has more functionality.
- ❖ **Scaling** — Redis supports master-slave load-splitting and sharding in clusters, enabling reliable scaling for large datasets and high loads. It automatically splits data among multiple nodes and continues operations during failures or communication issues within the cluster.

Redis - Comparison

Redis is designed primarily to work **with** a DBMS, not to **replace** one. Each has different strengths:

	Redis	RDBMS
Data Model	key-value store, where data is stored as simple key-value pairs	relational data model with tables, rows, and columns
Performance	optimised for fast read and write operations, making it ideal for caching and real-time analytics	offers more complex query capabilities, but not as fast as Redis for most read/write operations
Data Persistence	primarily uses in-memory data, with persistence only really used for disaster recovery purposes	provides durable storage by default, with data stored on disk and ACID compliance
Use Cases	caching, session management, real-time analytics, messaging queues	transaction processing, data warehousing, complex query operations

Redis - CRUD Operations

Redis can be complex to install and is **not needed** for the beginner databases covered in this module. Some basic commands are demonstrated here, but you do not need to run them locally.

Compared to other databases, Redis has a simpler implementation as it uses **read/write** operations instead of **CRUD** operations. [SET](#) is the basic command used to create or update a key-value pair in memory:

```
SET car "Audi"
```

SET in Redis: sets a value (Audi) to a **key** (car)



```
[Navit@10-140-24-166 ~ % redis-cli  
[127.0.0.1:6379> SET car "Audi"  
OK  
127.0.0.1:6379> ]
```

SET works with simple data types such as strings. You can find more examples using variations on the SET command for other data types in the [Redis documentation](#).

Redis - CRUD Operations

Read Operations

Retrieving a value is as simple as using the [GET command](#), using the same key that was used with SET.

GET in Redis: gets a value (Audi) using a key (car)

```
GET car
```



```
[127.0.0.1:6379> GET car  
"Audi"  
127.0.0.1:6379>
```

Typically the values stored in Redis would be more complex than this, allowing users to cache commonly accessed data from a database, calculations or perform analytics.

Redis - CRUD Operations

Delete Key

To remove a single key-value pair from your Redis data store, use the [DEL command](#) and specify the key:

```
DEL car
```

Drop Database

To remove all of the key-value pairs from your Redis data store, use the **FLUSHALL** command. This will remove all the data from Redis, so ensure that this is intended behaviour:

```
FLUSHALL
```

```
OK
[127.0.0.1:6379> GET car
(nil)
127.0.0.1:6379> █
```

Exercise 4

Using the logical database diagram from **Exercise 1**, show how Redis could be incorporated into your blogging system to improve performance.

- ❖ Add an extra box representing Redis to your existing diagram
- ❖ Use arrows to indicate the data flow between Redis and the database