What to Expect

Who is this course for and what are the learning outcomes?

WE'LL COVER THE FOLLOWING

Intended audience for this course

This course is designed for users who have no experience with database systems or structured Query Language (SQL). This course will introduce users to the different concepts and techniques that are used in designing a database management system.

image source = "https://community-cdn-digitalocean-com.global.ssl.fastly.net/assets/tutorials/images/large/Database-Mostov_v4.1_twitter-_-facebook.png?1549487063"

Learning outcomes

At the end of this course you will:

- Have a grasp of the basics of database systems.
- Be exposed to different types of databases.
- Learn about entity relationship diagrams and their uses.
- Be able to normalize databases in order to increase query efficiency.
- Have learned basic SQL commands to query the database.

So without further ado, let's get started.

Before the Advent of Databases

In this lesson we will discuss the history of databases and the system used before its inception.

WE'LL COVER THE FOLLOWING

Before the advent of database systems

The way computers manage data has come a long way over the last few decades. Today's users take for granted the many benefits found in a database system. However, it wasn't that long ago that computers relied on a much less elegant and costly approach to data management called the file-based system.

The File-based system

One way to keep information on a computer is to store it in permanent files. A company system has several application programs; each of them is designed to manipulate data files. These application programs have been written at the request of the users in the organization. The system just described is called the **file-based system**.

Consider the example of a university management system that uses the file-based system to manage the organization's data shown in the figure below:

Accounts

Student Information
• Name • Roll No. • Address • Fee vouchers

Department

Student Information
• Name • Roll No. • Address • Major • GPA

Hostel

Student Information
• Name • Roll No. • Address • Hostel no. • Room no.

University Management System

As we can see, data on the students is available to their respective departments, accounts section, hostel office, etc. Some of the data is common for all sections (like roll number, name, address and phone number of students). On the other hand, some of the data is exclusive to a particular section, such as hostel allotment number which is a part of the hostel office only.

Now you can imagine using that the file-based system to keep organizational information has several disadvantages which we will discuss in the next lesson.

Disadvantages of File-Based System

We will highlight the issues that arise when we use a file-based system and how databases are the solution.

WE'LL COVER THE FOLLOWING

Keeping in mind the university management system discussed in the previous lesson, we will shine a light on the problems that occur when we use a file-based system, which in turn are the reasons we shifted to database systems.

1. Data redundancy

Often, within an organization, files and applications are created by different programmers from various departments over long periods. This can lead to data redundancy, a situation that occurs when the same data is present in many places (files).

For example, if a student wants to change his/her phone number, he/she has to get it updated in various places (files). Similarly, old records must be deleted from all sections representing that student.

2. Data inconsistency

Data is said to be inconsistent if **multiple copies of the same data do not match with each other**, which wastes storage space and duplicates effort. Consider the case where a student's phone number is different in the accounts department and academics department, it will be inconsistent. Inconsistency may be because of typing errors or not updating all copies of the same data

may be because of typing errors or not updating all copies of the same data.

3. Difficult data access

Another issue is that the user needs to know the exact location of the file in order to access data, which can be a very cumbersome and tedious process. Let's say the user wants to know the hostel allotment number of a specific student from 10,000 unsorted students' records, it will prove to be quite monotonous.

4. Security problems

Using a file-based system may lead to **unauthorized access** of data. If a student gains access to the file that contains his marks, he can change it without authorization.

5. Difficult concurrent access

Concurrency is the ability of the database to allow multiple users access to the same information at the same time. Typically, in a file-based system, when an application opens a file, that file is locked. This means that **no one else can access the file at that moment in time**.

For example, if one of the departments in the university accesses the data on a specific student, the other departments will have to wait until the first department is done before they can access it. Thus, concurrency is not maintained in a file-based system.

The difficulties that arise from using the file-based system have prompted the development of a new approach to managing large amounts of organizational information called the *database approach*.

In the next chapter, we will dive deep into the fundamentals of database systems.

What is a Database?

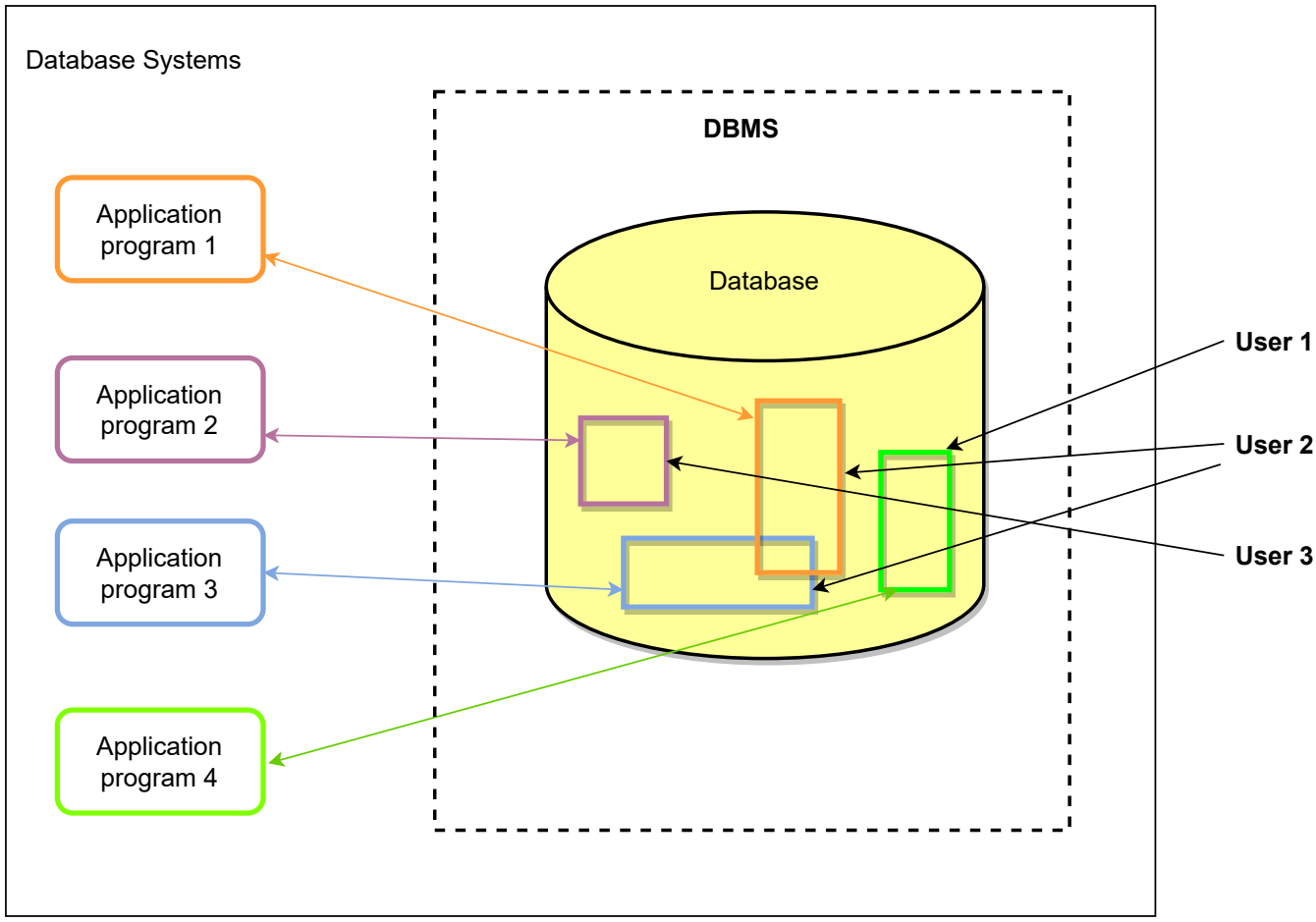
In this lesson we will discuss the basics of databases and their properties.

WE'LL COVER THE FOLLOWING

What is a database?

A database is a shared collection of related **data**. By **data,** we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know.

A database can be viewed as a repository of data that is defined once and then accessed by various users as shown in the figure below:



The application programs are nothing more than softwares that allow different users to send or retrieve data from the database.

Database properties

A database has the following properties:

- A database represents some aspect of the real world, sometimes called the mini-world or the **universe of discourse** (UoD). Changes to the mini-world are reflected in the database.
- A database is a **logically coherent collection of data** with some inherent meaning. A random assortment of data cannot correctly be called a database. For example, a list of random names cannot be considered a database but if we also note down those peoples' addresses and phone numbers, then the data will have some inherent meaning (i.e. it will be an address book).
- A database is designed, built, and populated with **data** for a **specific** purpose. It has an intended group of users and some preconceived applications in which these users are interested. For example, a

department in a university might be interested in gathering data regarding the students' GPA so that they can finalize the dean's honor list.

In other words, a database has some degree of interaction with events in the real world and an audience that is actively interested in its contents. The users of a database may perform some transaction or events may happen. For example, a student's marks are increased on an exam, which causes the information in the database to change. For the database to be considered reliable these changes must be reflected in the database as soon as possible, so it can remain a true reflection of the mini-world that it represents.

Examples of databases

A database can be of any size and complexity. Let's consider the previous example of the list of names and addresses. This database may consist of only a few hundred entries with simple text information.

On the other hand, consider a database maintained by a social media company such as Twitter, which has millions of users. The database has to maintain information on which users are related to one another in the form of *followers*, the content posted by each user, and a large number of other types of information needed for the correct operation of their website. For such websites, a large number of databases are needed to keep track of the constantly changing information required by the social media website.

In the next lesson, we will look at the basic concepts behind a Database Management System.

Database management systems (DBMS)

This lesson introduces DBMS and its functionality.

WE'LL COVER THE FOLLOWING

If an organization wants to adopt the database approach, then it needs a collection of programs that enable the users of the organization to create and maintain databases and control all access to them. This is achieved using a **database management system** (DBMS). The primary goal of a DBMS is to provide an environment that is both convenient and efficient for users to retrieve and store information.

Facilities provided by a DBMS

The DBMS is instrumental in facilitating the processes:

1. **Defining** a database involves defining the data types, structures, and constraints of the data to be stored in the database.
2. **Constructing** the database is the process of storing the data on a storage device that is controlled by the DBMS.
3. **Manipulating** a database involves querying the database to retrieve specific data, updating the database, etc.
4. **Sharing** a database allows multiple users and programs to access the database simultaneously.

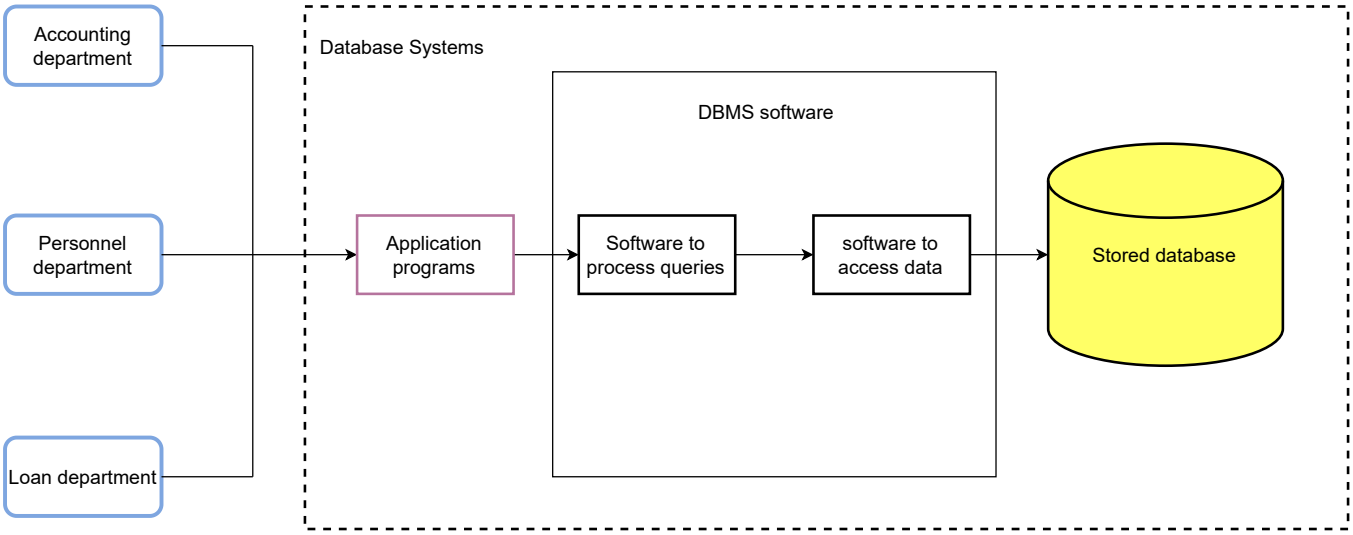
Other important terms

Furthermore, in order to access the database, we use **application programs** which send queries (requests) for data to the DBMS. A **query** causes some data to be retrieved.

To wrap up our discussion on basic terms, we call the database together with DBMS software a **database system**.

Example

With the database approach, we can have the traditional banking system as shown in the diagram below:



Database management system used by different departments in a bank

The individuals in the different departments use application programs to request the DBMS to retrieve the particular data that each individual is interested in. Only the application programs are visible to the end-user and they allow them to communicate with the DBMS. Then the DBMS, consisting of different software, fetches/stores the data from the database.

For example, an accountant in the accounting department wants information regarding the number of outstanding accounts. So he/she will send a request to the DBMS through the application software. After the DBMS retrieves the information from the database, the results will be displayed through the application program.

In the next lesson, we will take a look at an example of a database: the university database.

An Example of a Database

In this lesson we will take a look at a simple university database.

WE'LL COVER THE FOLLOWING

- A university database example

Before we move onto the example, we must identify the need to store data in tabular form.

One of the main reasons to store data regarding a specific object in a table is ease of understanding. So, instead of keeping relevant information regarding an object (like a student) in separate files, tables allow us to keep the important data regarding that object (like student ID, name, address, age, etc.) in one place.

A **table** is a collection of related data held in a table format within a database. It consists of columns and rows.

Column_1	Column_2	Column_3
Data_Item1	Data_Item2	Data_Item3
Data_Item4	Data_Item5	Data_Item6
...	...	...

Each row in a relational database represents one instance of the type of object described in that table. A row is also called a **record**. While the columns in a table are the set of facts that we keep track of regarding that type of object. A column is also called an **attribute**.

A university database example

Let's consider a university database that is used to maintain information concerning students, courses, and departments in a university environment. The diagram below shows the database structure and a few sample data records.

Student Table

ID	First_Name	Last_Name	Class	Major
2001	Adam	Smith	Junior	CS
2342	Jonathan	Joestar	Sophomore	Economics
2343	Lucas	Klein	Senior	Physics

Course Table

Course_ID	Course_Name	Course_credits
CS200	Intro to programming	4
MATH100	Calculus-I	3
CS300	Advanced Programming	3

Department Table

Department_Code	Department_Name
1	Computer Science

2	Electrical Engineering
3	Physics
4	Biology

Instructor Table

Instructor_ID	Instructor_fname	Department_Code
12	Sam	1
22	Tom	2
04	David	3

Grade Table

ID	Course_ID	Grade
2001	CS200	A-
2220	EE100	C
2343	PHY220	B

This database is comprised of five tables, each of which stores **data records** of the same type. The STUDENT table stores data on each student like the student's name, identification (ID) number, etc. While the COURSE table contains information about each particular course.

To properly define this database, we must specify the structure of the rows of each table by specifying the different types of **data elements** to be stored in each row. In the above diagram, each row in the STUDENT table includes data

that represent the name, ID, major, etc. of a single student only. Similarly, each

row in the COURSE table includes data that represent the name, ID, and credit hours of a single course.

We also notice that the data in each column is of a specific type. For example, we observe that the name of a student in the STUDENT table is a string of alphabetic characters, while the ID number of a student is an integer, and so on.

Furthermore, to construct the university database, we store data to represent each student, course, instructor, and department as a row in the appropriate table. It is important to note that some of the records in different tables are related to each other. For example, we can see in the INSTRUCTOR table that the first instructor, Sam, is from the computer science department as the *Department_Code* is common between the two tables.

Finally, after we have defined and constructed the database we can use the DBMS to retrieve specific rows from the different tables. For example, we can retrieve the names of professors that belong to a particular department.

In the next lesson, we will dive deep into the characteristics of databases.

Characteristics of the Database Approach

In this lesson we will discuss the characteristics of a database.

WE'LL COVER THE FOLLOWING

Characteristics of a database

Several characteristics distinguish the database approach from the file-based system. These include:

1. Self-describing nature of a database system

A database system is self-describing because it not only contains the database itself, but also the **meta-data** which defines and describes the data and relationships between tables in the database. This information is stored in a catalog by the DBMS software. The separation of data and information about the data makes the database system different from the traditional file-based system in which the data definition is part of the application programs.

Let's consider the university database example in the previous lesson. The DBMS catalog will store the definitions of all the tables in that database. The figure below shows part of the database catalog:

Tables

Table_name	Number_columns

Student	3
Course	3
Department	2
Instructor	3

Columns

Attributes	Datatype	Belongs_to
ID	Integer (4)	Student
First_Name	Character (50)	Student
Last_Name	Character (50)	Student
Course_name	Character (50)	Course
Course_Section	Integer (4)	Course
Department_Name	Character (50)	Department
...	...	...

Now consider the scenario in which a user wants to access the information of a student, i.e., the student personal information(ID, first name, and last name), the department to which the student belongs, etc. Although the information is stored in separate tables, the DBMS software will refer to the catalog in order to determine the structure of these tables as well as the relationships between them. In this way, the DBMS will retrieve the information relevant to the user.

2. Insulation between program and data

In the file-based system, the structure of the data files is defined in the application programs so if a user wants to change the structure of a file, all the programs that access that file might need to be changed as well.

On the other hand, in the database approach, the data structure is stored in the system catalog and not in the programs. Therefore, one change is all that is needed to change the structure of a file. This insulation between the programs and data is also called **program-data independence**.

For example, we have an application program that is designed to work with the current structure of the STUDENT table. If we want to add another piece of data to the STUDENT table, say a column called `Home_Address`, such a program will no longer work and would have to be changed.

On the other hand, in a DBMS environment, we only need to change the description of the STUDENT table in the catalog to reflect the inclusion of the new data item `Home_Address` in each student record. So the next time a DBMS program refers to the catalog, the new structure of the STUDENT table will be used.

3. Support for multiple views of data

A database supports multiple views of data. A **view** is a subset of the database, which is defined and dedicated for particular users of the system. Multiple users in the system might have different views of the system. Each view might contain only the data of interest to a user or group of users. For example, one user of the university database may be interested only in accessing the names of the professors in each department. The view for this user is shown below:

Department_code	Department_name	Instructor_name
1	Computer Science	Sam Winchester Doug Waterman Glen Orsburn Dean Wright

		Dean Wright
2	Electrical Engineering	Terri Keane James Avery

The user will only be presented with the relevant information rather than the whole database.

4. Sharing data and multiuser system

Current database systems are designed for multiple users. That is, they allow many users to access the same database at the same time. This access is achieved through features called **concurrency control strategies**. These strategies ensure that the data accessed are always correct and that data integrity is maintained. Consider the case when multiple reservation agents try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one agent at a time so that the same seat is not assigned to multiple passengers.

In the next lesson, we will go over the benefits of using the database approach.

Benefits of the Database Approach

In this lesson we will discuss the advantages of the database approach.

WE'LL COVER THE FOLLOWING

Benefits of using the database approach

We will now discuss some advantages of using a database system and the capabilities that a good DBMS should possess:

1. Control of data redundancy

In the database approach, ideally, each data item is stored in only one place in the database. This is known as **data normalization**, and it ensures consistency and saves storage space. In some cases, data redundancy still exists to improve system performance but it is kept to a minimum.

2. Data sharing

The integration of all the data for an organization within a database system has many advantages. First, it allows for data sharing among employees and others who have access to the system. Second, it gives users the ability to generate more information from a given amount of data than would be possible without the integration.

3. Enforcement of integrity constraints

Database management systems must be able to define and enforce certain constraints to ensure that users enter valid information and maintain data integrity. A database constraint is a restriction or rule that dictates what can be entered or edited in a table, such as adding a valid course name in the `Course_Name` column in the COURSE table.

There are many types of database constraints. One of them is data type, which determines the sort of data that can be entered. For example, integers only. Another restraint is **data uniqueness**, which specifies that data item values must be unique, such as every record in the STUDENT table must have a unique value for `ID`.

4. Restriction of unauthorized access

Not all users of a database system will have the same access privileges. For example, one user might have **read-only** access (i.e., the ability to read a file but not make changes), while another might have **read and write privileges** (which is the ability to both read and modify a file). For this reason, a database management system should provide a security subsystem to create and control different types of user accounts and restrict unauthorized access.

5. Backup and recovery facilities

Backup and recovery are methods that allow you to protect your data from loss. The database system provides a facility for backing up and recovering data. If a hard drive fails and the database stored on the hard drive is not accessible, the only way to recover the database is with a backup.

If a computer system fails in the middle of a complex update process, the recovery subsystem is responsible for making sure that the database is restored to its original state.

The next lesson will include a quick quiz to test your knowledge of these fundamentals.

Quiz!

This quiz will test your knowledge of the fundamentals of databases.

1 Which of the following is NOT an example of a database?

COMPLETED 0%

1 of 5

With this quiz, the chapter comes to an end. This chapter introduced us to the fundamentals characteristics of databases with some new examples.

In the next chapter, we will focus on the first step in database design i.e. data modeling.

Introduction to Data Models

We will look at the meaning behind data models and the different types of models.

WE'LL COVER THE FOLLOWING

In order to store data in a database system, we need some data-structures. Hence the database systems we use normally include some complex data structures which we normally do not use. To make the system efficient in terms of data retrieval, and reduce complexity in terms of usability, developers use **data abstraction** i.e., hide irrelevant details from the users.

In order to achieve this abstraction, we use **data models**.

> A data model is a collection of concepts or notations for describing data, data relationships, data semantics, and data constraints.

We will highlight what these terms mean in the next few lessons.

Most data models also include a set of basic operations for manipulating data in the database.

Types of data models

The different types of data models can be classified into the following categories:

1. High-level conceptual data models

High-level conceptual data models provide a way to present data that is similar to how people perceive data. A typical example is an entity-relationship model, which uses concepts like entities, attributes, and relationships.

Entity relationship model

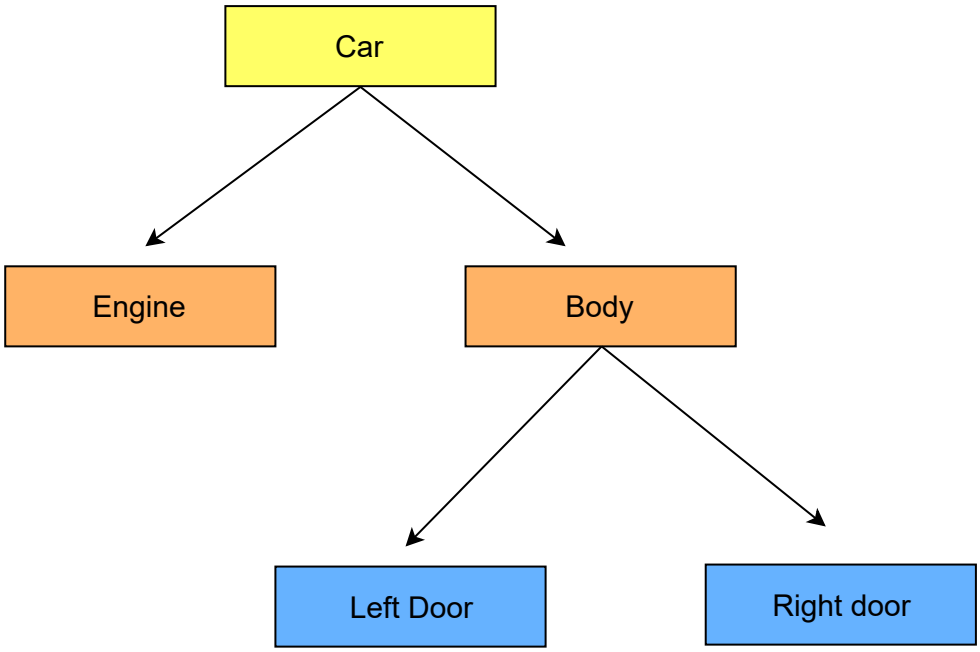
An **entity** represents a real-world object such as an employee or a project. The entity has **attributes** that represent properties such as an employee's name, address, and birthdate. A **relationship** represents an association among entities; for example, an employee works on many projects. A relationship exists between the employee and each project.

2. Record-based logical data models

Record-based logical data models provide concepts users can understand but are still similar to the way data is stored on the computer. Three well-known data models of this type are: hierarchical, network, and relational data models.

Hierarchical model

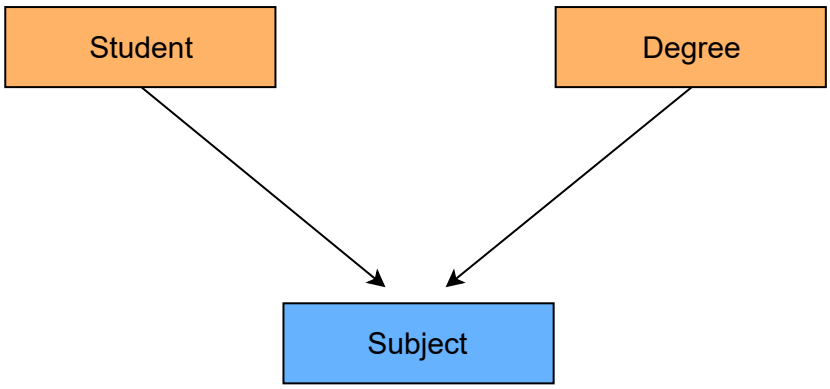
In a **hierarchical model**, data is organized into a tree-like structure, implying a single parent for each record. This structure mandates that each child record has only one parent, whereas each parent record can have one or more child records. This concept is illustrated below:



Car is the parent of both Engine and Body, so each child has only one parent.

Network model

The **network model** expands upon the hierarchical structure, allowing each record to have multiple parent and child records, forming a generalized graph structure. It was the most popular model before being replaced by the relational model. The network model is shown in the diagram below:



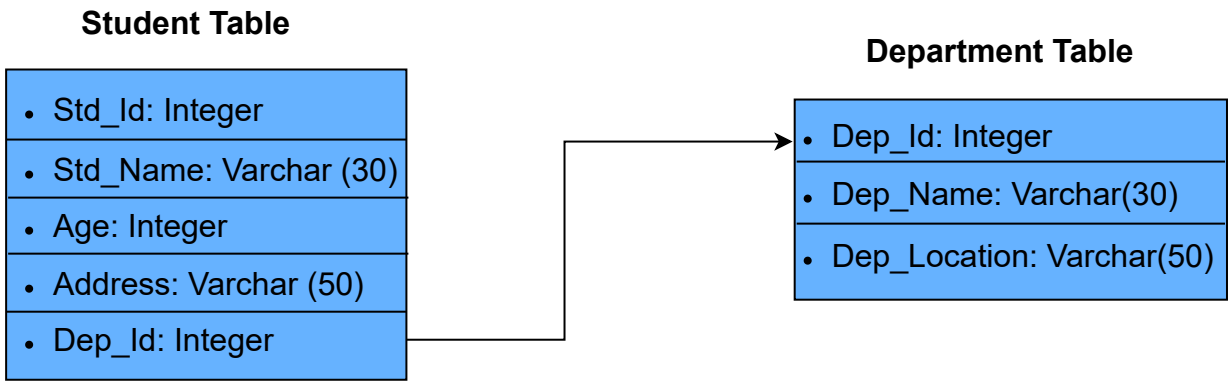
In this figure, we can see that the Subject is the child of Student and Degree. So, it has two parent classes.

Relational model

The **relational model** represents data as relations or tables. For example, the university database system contains multiple tables (relations) which in turn have several attributes (columns) and tuples (rows).

3. Physical data models

The physical data model represents how data is stored in computer memory, how it is scattered and ordered in the memory, and how it would be retrieved from memory. Basically physical data model represents each table, its columns, and specifications, etc. It also highlights how tables are built and related to each other in the database. The diagrammatic representation of physical models is shown below:



The STUDENT table is related to the DEPARTMENT table through the Dep_Id attribute.

As we can see the STUDENT table consists of attributes like `Std_Id`, `Std_Name`, `Age` etc. along with their data type. Same for the DEPARTMENT table. The arrow simply shows how these two tables are connected in this model.

In the next lesson, we will discuss schemas and instances.

Schemas and Instances

The schema and instance are the essential terms related to databases. In this lesson, we will discuss the key differences between the two.

WE'LL COVER THE FOLLOWING

Database schema

A schema is the blueprint of a database. The names of tables, columns of each table, datatype, functions, and other objects are included in the schema.

We use the **schema diagram** to display the schema of a database. The schema diagram for the university database can be seen below:

Student Table

ID	First_Name	Last_Name	Class	Major

Course Table

Course_ID	Course_Name	Course_credits

Department Table

Department_Code	Department_Name

Instructor Table

Instructor_ID	Instructor_fname	Department_Code

Grade Table

ID	Course_ID	Grade

> It is important to note that the schema diagram is not the same thing as the schema.

The schema diagram displays only **certain aspects** of a schema, such as the names of record types and data items. Other features (although present in the schema) are not specified in the schema diagram; for example, the above diagram shows neither the data type of each item nor the relationships among the various tables.

On another note, the schema is not changed frequently, sometimes changes need to be applied to the schema as the requirements of the application change. For example, we may decide to add another data item to each record in a table, such as adding an `Address` field to the Student schema. This schema modification or alteration is known as **schema revolution**.

Database instance

An instance is the information collected in a database at some specific moment in time, also known as the **database state**. It is a snapshot of the current state or occurrence of a database. Each time data is inserted into or deleted from the database, it changes the state of the database. That is the reason why an instance of the database changes more often.

The starting state of the database is acquired when the database is first loaded with initial data. From then onwards, each time data is updated we get a new database instance. At any point in time, there is a current state associated with a database.

To explain the concept of instances further we will look at a snapshot of the STUDENT table at a particular moment in time.

ID	First_Name	Last_Name	Class	Major
1001	Bob	Dylan	Junior	Maths
1002	Ceaser	Zappelli	Freshman	Economics
1003	Antony	Rodgers	Senior	Psychology
1004	George	Miller	Sophomore	Computer science
...	...	...	...	...

This is just one instance of the STUDENT table. If we add, remove or update records into this table than we will enter a new database state.

In the next lesson, we will dive deep into the three-schema architecture.

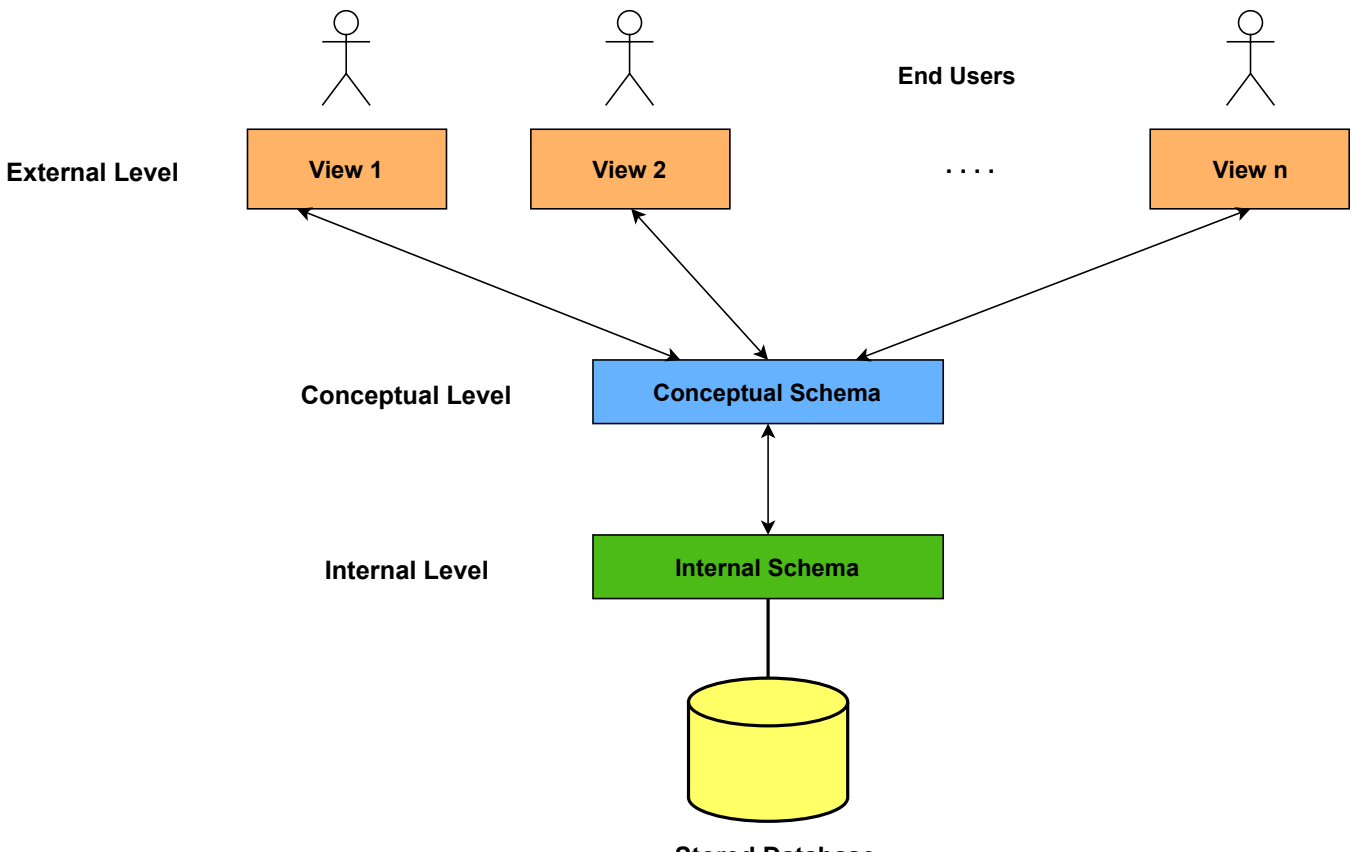
The Three-Schema Architecture

In this lesson, we will define the three levels of data abstraction.

WE'LL COVER THE FOLLOWING

The three-schema architecture

The goal of the three-schema architecture is to separate the user applications from the physical database.



Stored Database

Three-Schema Architecture

In this architecture, schemas can be defined at the following three levels:

1. External schema

An external schema describes the part of the database that a specific user is interested in. It hides the unrelated details of the database from the user like the exact process of retrieving or storing data from the database. There is a different external view for each user of the database.

An external view is just the content of the database as it is seen by one particular user. For example, a user from the sales department will only see sales-related data.

2. Conceptual schema

The conceptual schema describes the database structure of the whole database for the community of users. This schema hides information about the physical storage structures and focuses on describing data types, entities, relationships, etc. Usually, a record-based logical data model is used to describe the conceptual schema when a database system is implemented.

3. Internal schema

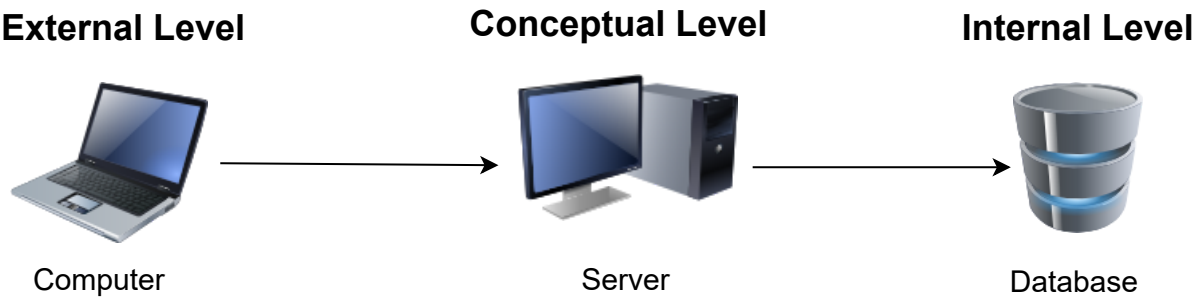
The internal schema describes how the database is stored on physical storage devices such as hard drives. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.

Example of the three-schema architecture

To make things clearer, consider the example of visiting a website through your personal computer.

The web browser on your computer is the external level as you only receive the final web page displayed on your screen without knowing what goes on in the background. The server hosting the website is the conceptual level as it receives your request, retrieves the data you want from the database, and then sends that data back to your computer. The database(stored on some physical media) represents the physical level. It houses the data you are

interested in.



Why use the three-schema architecture

Here are some of the goals that can be achieved using the three schema architecture:

- Every user should be able to access the same data, but be able to see a customized view of the data.
- The user does not need to deal directly with physical database storage details.
- The database administrator should be able to change the database storage structure without disturbing the user's view.
- The internal structure of the database should remain unaffected when changes are made to the physical aspects of storage.

We will move onto the concept of data independence in the next lesson.

Data Independence

In this lesson, we will look at one of the results of the three-schema architecture.

WE'LL COVER THE FOLLOWING

Data independence is defined as a property of the database management system that helps to change the database schema at one level of a database system without changing the schema at the next highest level. The three-schema architecture can be used to further explain the concept of data independence. We outline the two types of data independence:

1. Logical data independence

The ability to change the conceptual schema without changing the external schema or user view is called logical data independence. For example, the addition or removal of new entities, attributes or relationships to this conceptual schema should be possible without having to change existing external schemas or rewriting existing application programs.

In other words, changes to the conceptual schema (e.g., alterations to the structure of the database, like adding a column or other tables) should not affect the function of the application (external views).

2. Physical data independence

Physical data independence helps you to separate the conceptual schema from the internal schema. It allows you to provide a logical description of the database without the need to specify physical structures.

So for example, there is a change to the internal schema because some physical files were reorganized, for example, by creating additional access structures to improve the performance of retrieval or updates. Physical data independence makes sure that these changes to the internal schema do not affect the conceptual schema. In turn, the external schemas need not be changed as well.

Does data independence work in reality?

Generally, physical data independence exists in most databases and file environments where physical details, such as the exact location of data on disk or the type of storage device, are hidden from the user. On the other hand, logical data independence is harder to achieve because it must accommodate changes in the structure of the database without affecting application programs; which is a much stricter requirement.

In the next lesson, we will outline the different classifications of database management systems.

Classification of Database Management Systems

This lesson describes the different metrics by which we can classify DBMS.

WE'LL COVER THE FOLLOWING

Database management systems can be classified based on several criteria, such as the data model, user numbers and database distribution, each of which is described below.

Classification based on data model

The most popular data model in use today is the relational data model. Well-known DBMSs like Oracle, MS SQL Server, DB2 and MySQL support this model. Other traditional models, such as hierarchical data models and network data models, are still used in the industry mainly on mainframe platforms. However, they are not commonly used due to their complexity. These are all referred to as traditional models because they preceded the relational model.

In recent years, the newer **object-oriented data models** were introduced. This model is a database management system in which information is represented in the form of objects as used in object-oriented programming. Object-oriented databases are different from relational databases, which are table-oriented. Object-oriented database management systems (OODBMS)

combine database capabilities with object-oriented programming language capabilities.

The object-oriented models have not caught on as expected, so they are not in widespread use. Some examples of object-oriented DBMSs are O2, ObjectStore, and Jasmine.

Classification based on number of users

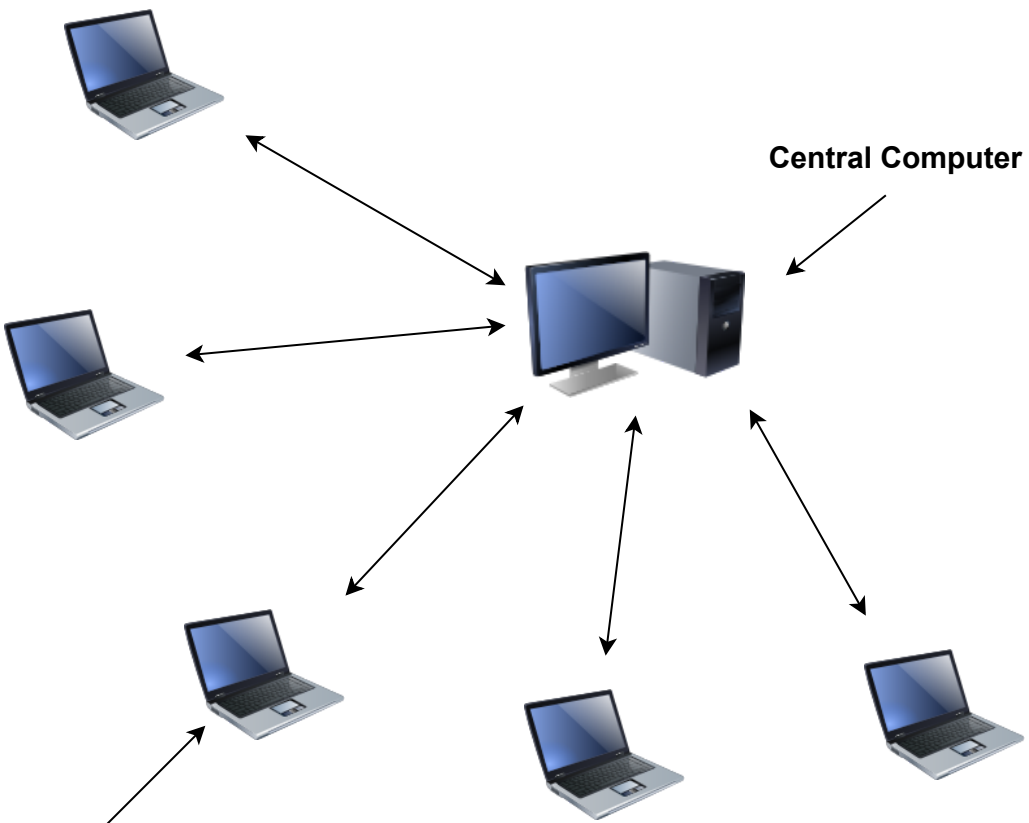
A DBMS can be classified based on the number of users it supports. It can be a **single-user database system**, which supports one user at a time, or a **multi-user database system**, which supports multiple users concurrently.

Classification based on database distribution

There are four main distribution systems for database systems and these, in turn, can be used to classify the DBMS.

Centralized systems

Within a centralized database system, DBMS and database are central. i.e., stored at a single location and is used by several other systems. This is illustrated below:

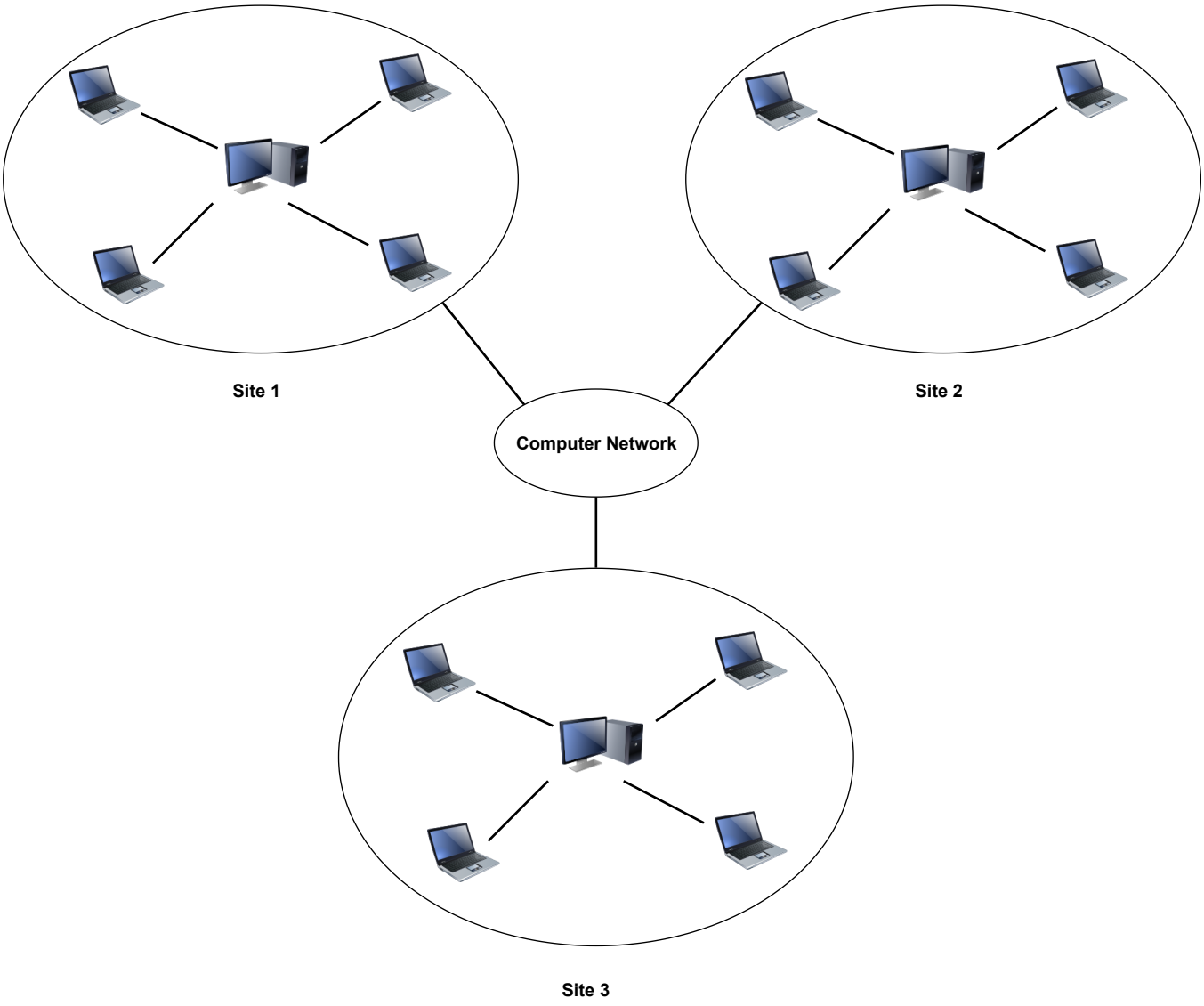


Workstation or terminal

The database is stored in the central computer. The different users can access the database using their terminals.

Distributed database system

In a distributed database system, the actual database and the DBMS software are distributed across various sites that are connected by a computer network, as shown in the figure below:



The database is distributed between the three sites

Homogeneous distributed database systems

Homogeneous distributed database systems use the same DBMS software from multiple sites. Data exchanged between these various sites can be handled easily. For example, library information systems by the same vendor, such as Geac Computer Corporation, use the same DBMS software which allows easy data exchange between the various Geac library sites.

Heterogeneous distributed database systems

In a heterogeneous distributed database system, different sites might use different DBMS software, but there is additional common software to support data exchange between these sites. For example, the various library database systems use the same machine-readable cataloging (MARC) format to support library record data exchange.

Quiz time! The next lesson will test your knowledge of data models.

Quiz!

This quiz will test your knowledge of data models.

1 Which of the following is NOT a record-based logical data model?

COMPLETED 0%

1 of 7

We have reached the end of this chapter. Good Job! At this point, we should be familiar with the first step in database design: schemas. You also learned to classify databases by their different attributes.

In the next chapter, we will learn all there is to know about the entity-relationship data model.

Intro to Entity-Relationship Model

This lesson introduces the concept of entity-relationship model.

WE'LL COVER THE FOLLOWING

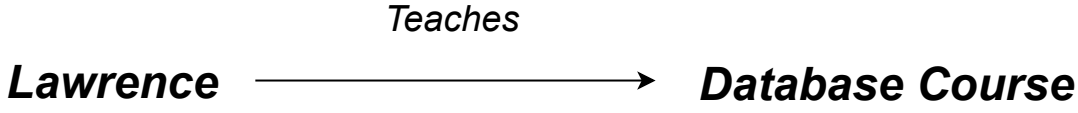
The entity-relationship data model

The entity-relationship (ER) data model is a high-level conceptual data model that has existed for over 35 years. It is well suited to data modeling for use with databases because it is fairly abstract and is easy to discuss and explain. ER models, also called ER schemas, are represented by **ER diagrams**.

ER modeling is based on two concepts:

- **Entities**, defined as tables that hold specific information (data).
- **Relationships**, defined as the associations or interactions between entities.

Here is an example of how these two concepts might be combined in an ER data model: Prof. Lawrence (entity) teaches (relationship) the database systems course (entity).

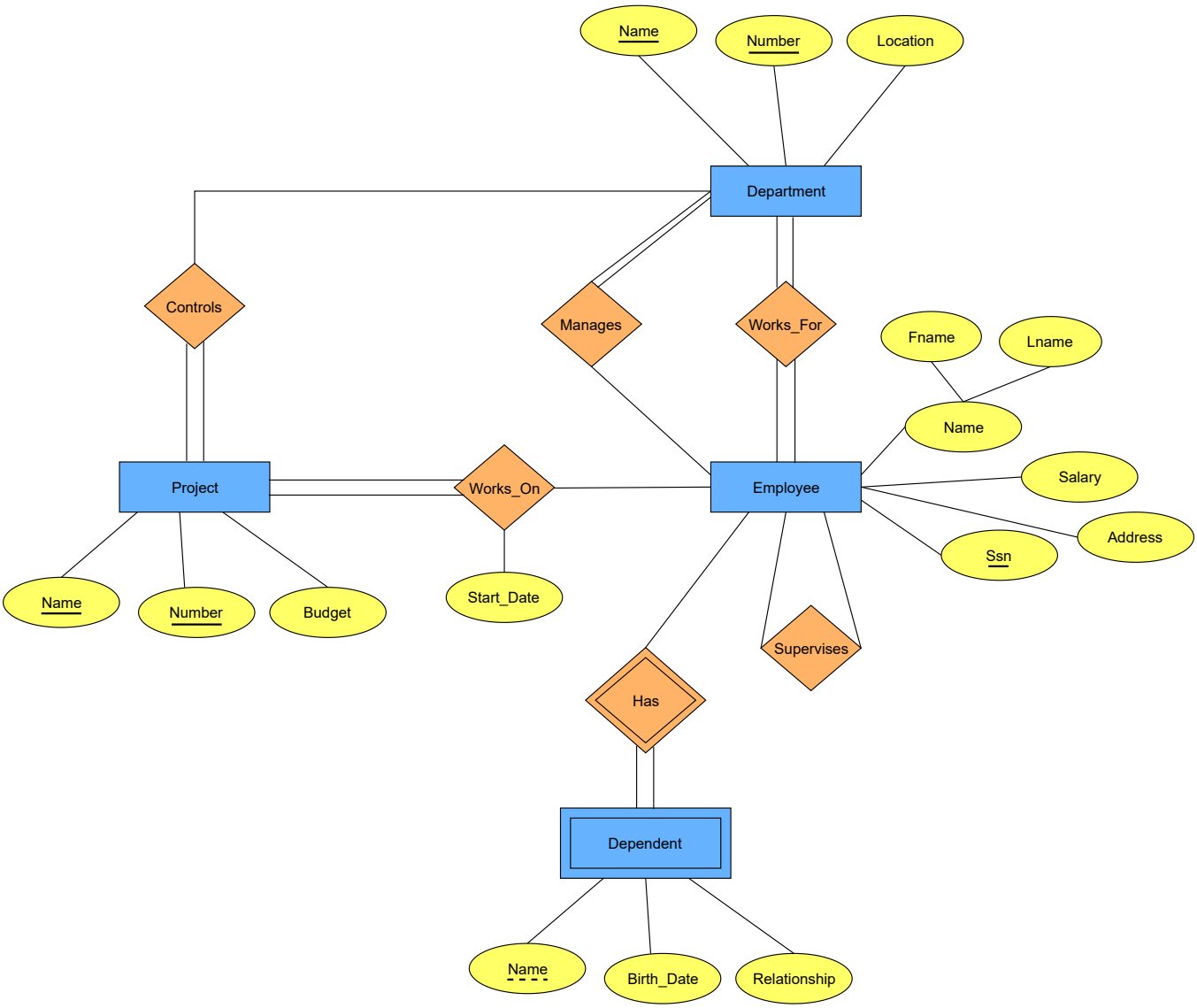


Example: a company database

For the rest of this chapter, we will use a sample database called the COMPANY database to illustrate the concepts of the ER model. This database contains information about employees, departments, and projects. Important

points to note include:

- There are several departments in the company. Each department has a unique identification number, a name, location of the office and a particular employee who manages the department.
- A department controls several projects, each of which has a unique name, a unique number, and a budget.
- Each employee has a name, identification number, address, salary, and birthdate. An employee is assigned to one department but can work on several projects. We need to record the start date of the employee in each project. We also need to know the direct supervisor of each employee.
- We want to keep track of the dependents for each employee. Each dependent has a name, birth date and relationship with the employee.



An ER diagram of a company database

The above figure shows how the schema for this database application can be displayed employing the ER diagram. This figure will be explained gradually

displayed employing the ER diagram. This figure will be explained gradually

as the ER model concepts are presented in the next few lessons so don’t worry about it too much.

In the next lesson, we will look at one of the basic components of an ER diagram: entities.

Entities, Entity Sets and Entity Types

In this lesson, we will discuss the first component of an ER diagram: entities.

WE'LL COVER THE FOLLOWING

What is an entity?

An entity is an object in the real world with an independent existence that can be differentiated from other objects. An entity might be:

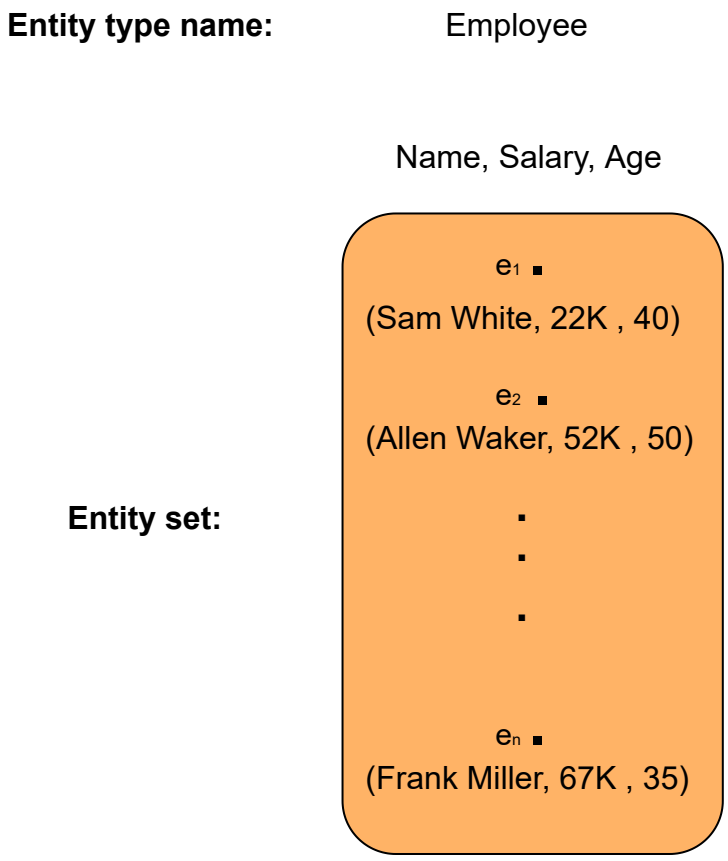
- An object with physical existence (e.g., a lecturer, a student, a car).
- An object with conceptual existence (e.g., a course, a job, a position).

Each entity has **attributes** which are the particular properties that describe it. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, etc. A particular entity will have a value for each of its attributes (e.g. an employee named Steve, who is 23 years old, lives in Ohio, and earns $50,000, etc.). The attribute values that describe each entity become a major part of the data stored in the database.

Entity types and entity sets

A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of their employees. These EMPLOYEE entities share the same attributes (like `Name`, `Salary`, `Age`), but each entity has its own values for each attribute. An **entity type** defines a collection (or set) of entities that have the same attributes. Each entity type in the database is described by its name and attributes.

The figure below shows the EMPLOYEE entity type as well as a list of some of the attributes for that type. A few individual entities are also illustrated, along with the values of their attributes.

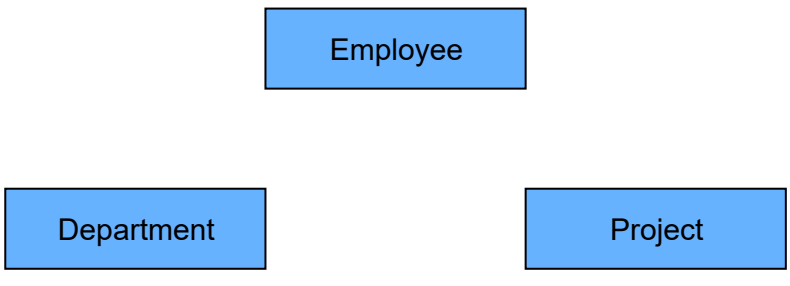


Example of entity type and entity set

The collection of all entities of a particular entity type in the database at any point in time is called an **entity set**. The entity set is usually referred to using the same name as the entity type, even though they are two separate concepts.

Representation in ER diagrams

An entity type is represented in ER diagrams as a rectangular box enclosing the name of the entity type. For example we can see the entities from the previous lesson below:



Representation of entities in ER model

Representation of entities in ER model

In the next lesson, we will expand the concept of attributes and highlight the different types of attributes in an ER diagram.

Attributes

In this lesson, we discuss the different types of attributes present in an ER diagram.

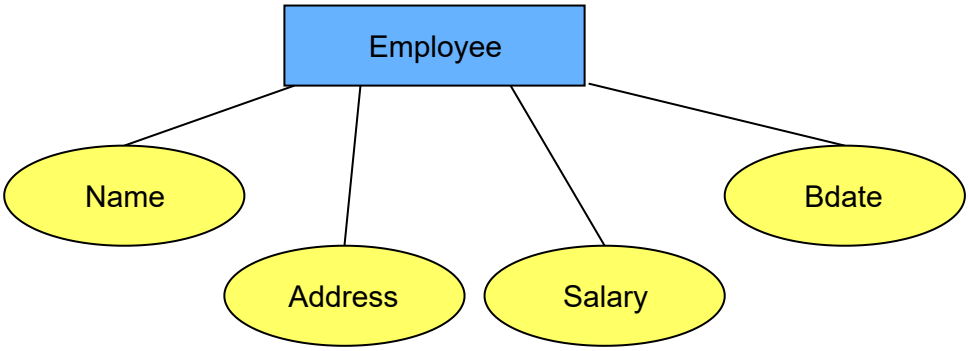
WE'LL COVER THE FOLLOWING

Attributes

As stated in the previous lesson, each entity is described by a set of attributes. For example, the EMPLOYEE entity has attributes `Name`, `Address`, and `Salary`, etc.

Each attribute has a name, is associated with an entity, and has a range of values it is allowed to take (an employee's salary cannot be negative). However, information about the domain of an attribute is not presented in the ER diagram.

In the entity-relationship diagram shown below, each attribute is represented by an oval with a name inside.



An EMPLOYEE entity represented along with its attributes.

Types of attributes

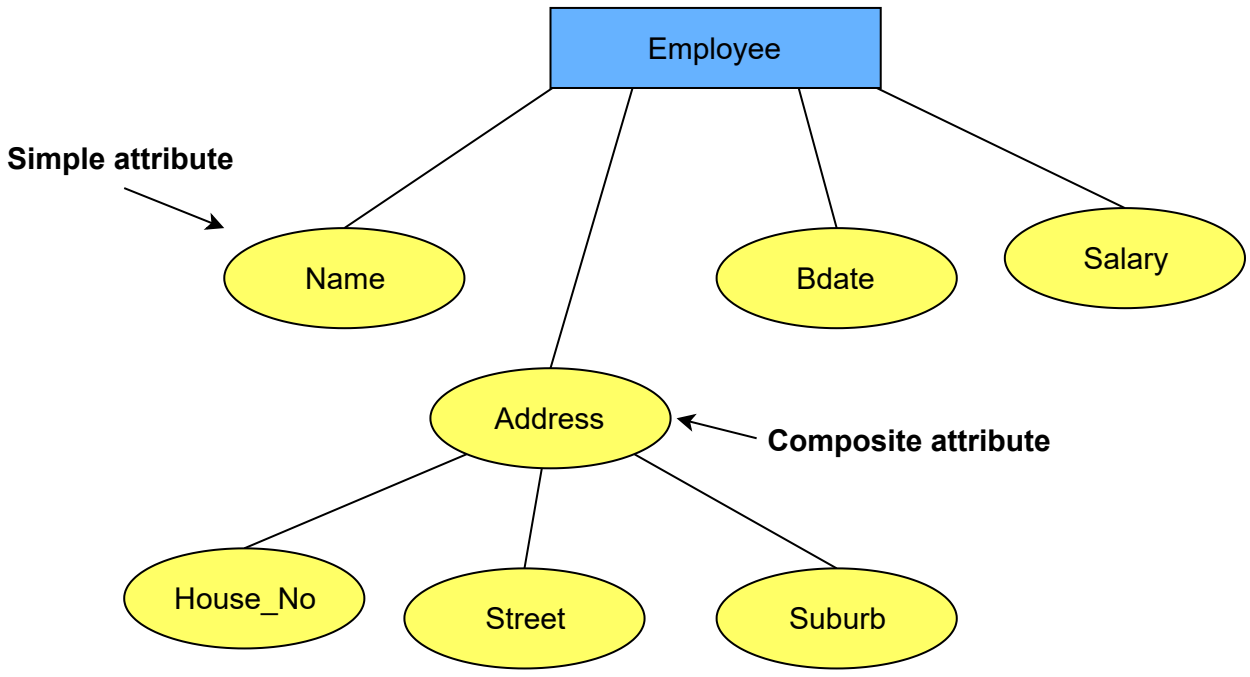
There are a few types of attributes you need to be familiar with.

Simple attributes

Simple attributes are the atomic value, i.e., they cannot be further divided. They are also called single-value attributes. In the COMPANY database, an example of this would be: `Name` = 'John' and `Age` = 23.

Composite attributes

Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings. Therefore, composite attributes consist of a hierarchy of attributes. Using our COMPANY database example, the `Address` attribute may consist of `House_No`, `Street` and `Suburb`. So this would be written as Address = 59 (`House_No`), Meek Street (`Street`), Kingsford (`Suburb`). The diagram below illustrates this concept:

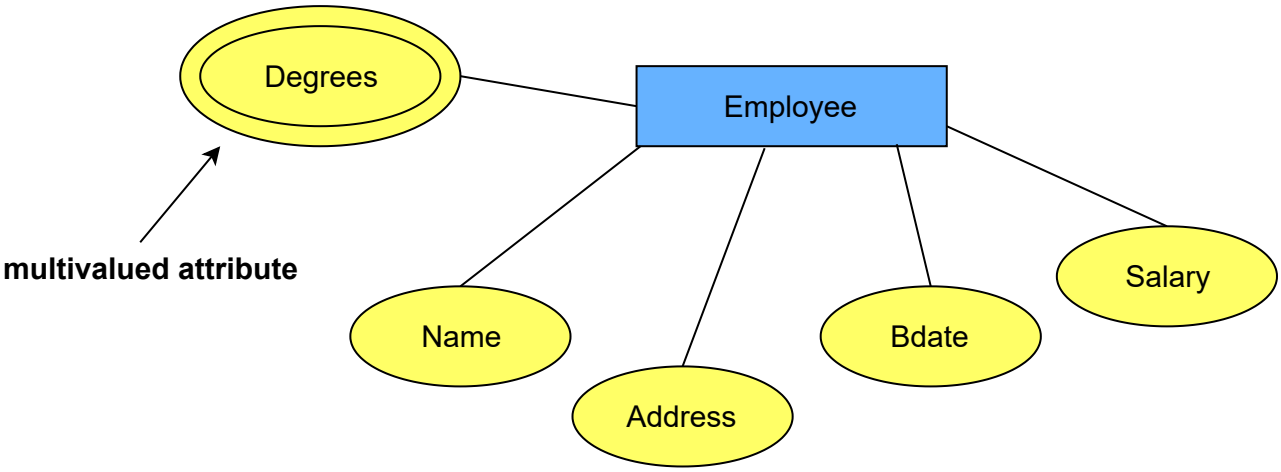


We can see simple and composite attributes highlighted in the diagram.

Multivalued attributes

Multivalued attributes have a set of values for each entity. An example of a multivalued attribute from the COMPANY database; one employee may not have any college degrees, another may have one, and a third person may have

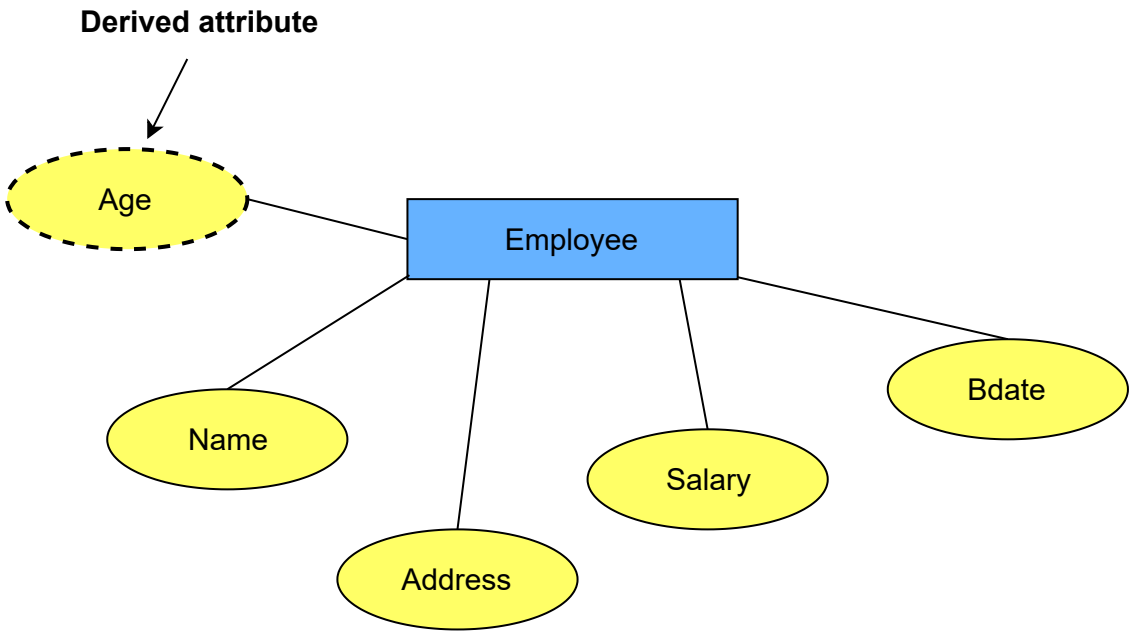
two or more degrees. Therefore, different people can have a different amount of values for the `Degrees` attribute. This is illustrated in the diagram below:



In ER diagrams multivalued attributes are represented with double outlines.

Derived attributes

Derived attributes are attributes that contain values calculated from other attributes. An example from the COMPANY database is that `Age` can be derived from the attribute `Bdate`. In this situation, `Bdate` is called a **stored attribute**, which is physically saved to the database. The image below highlights this concept:



In ER diagrams multivalued attributes are represented with dotted outlines.

The next lesson will be about keys and how they act as a constraint for entities.

Keys

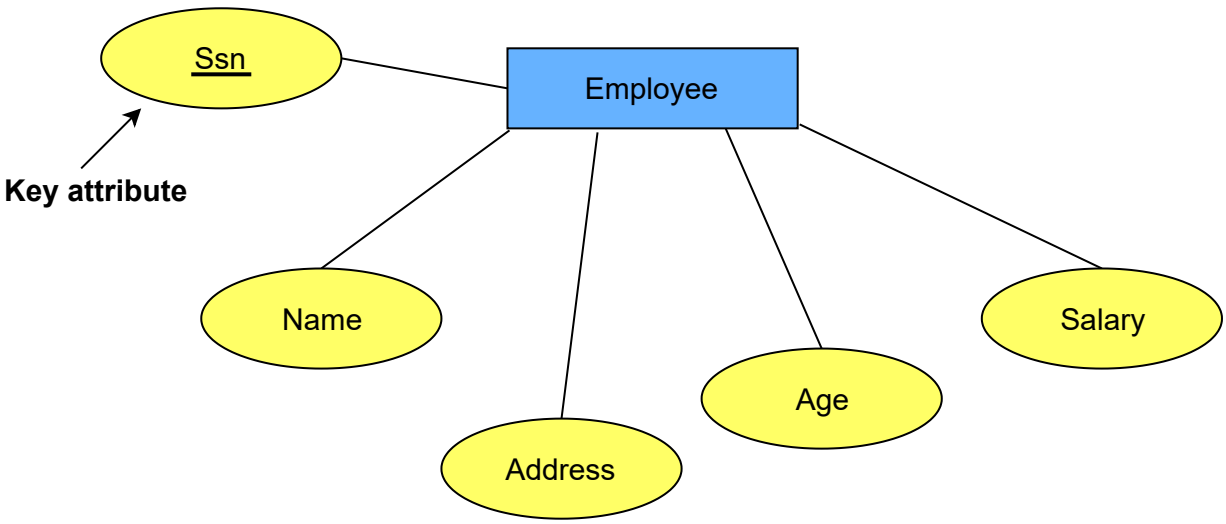
In this lesson, we discuss the basics of keys and how they are represented using ER diagrams.

WE'LL COVER THE FOLLOWING

Key attributes of an entity type

An important constraint on the entities of an entity type is the **key** or **uniqueness** constraint on attributes. An entity type usually has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a **key attribute**, and its values can be used to identify each entity uniquely. For the EMPLOYEE entity type, a typical key attribute is `Ssn` (Social Security number) as each person has a unique social security number.

In an ER diagram, each key attribute has its name underlined inside the oval, as illustrated in the figure below:



Ssn can be used to uniquely identify each employee so it is the key attribute.

Specifying that an attribute is a key of an entity type means that the preceding uniqueness property must hold for every entity set of the entity type. Hence, it is a constraint that prohibits any two entities from having the same value for the key attribute at the same time. This unique attribute is also known as the **primary key**.

Other examples of primary keys in the COMPANY database include `Dept_Id` which can be used to identify each department in the company. Also, `Project_Id` acts as the key attribute for the PROJECT entity.

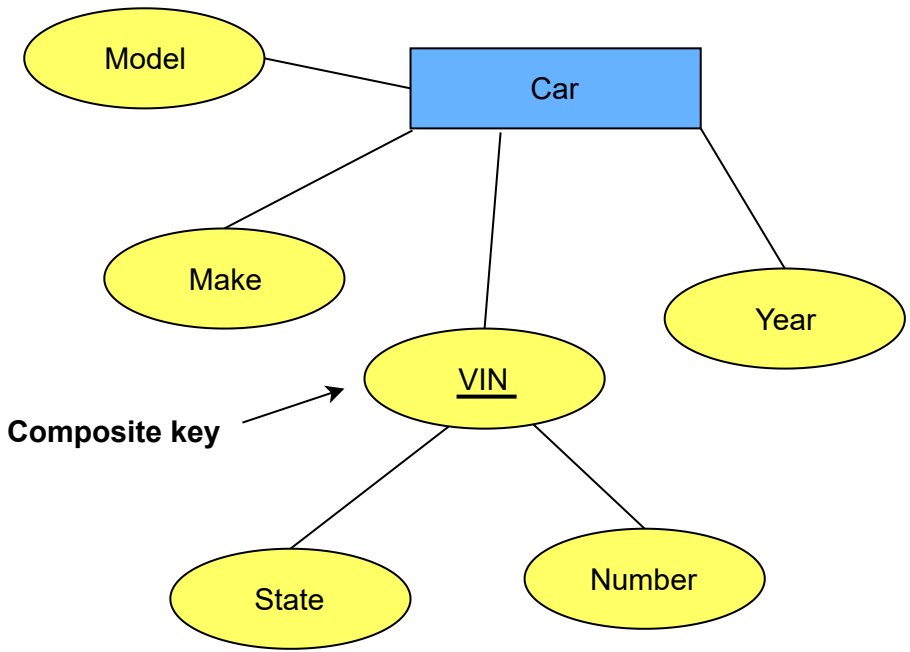
Composite keys

Sometimes a single attribute is not enough to uniquely identify each entity within an entity set.

Let's take the example of a CAR entity. The `VIN` (vehicle identification number) attribute is formed from two simple component attributes, `State` and `Number`, neither of which is a key on its own, as both can never be unique for every car. However, their combination can uniquely identify every car in the country.

In cases like this, several attributes together form a key, meaning that the combination of the attribute values must be distinct for each entity. If a set of attributes possesses this property, the proper way to represent this in the ER model that we describe here is to define a composite attribute and designate it as a key attribute of the entity type. This is called a **composite key**.

So in the example of the CAR entity `VIN` is a composite key.



VIN is a composite key made up of State and Number

In the next lesson, we will move onto the next component in an ER diagram: relationships.

Relationships, Relationship Sets and Relationship Types

In this lesson, we will study the basics of relationships between entities.

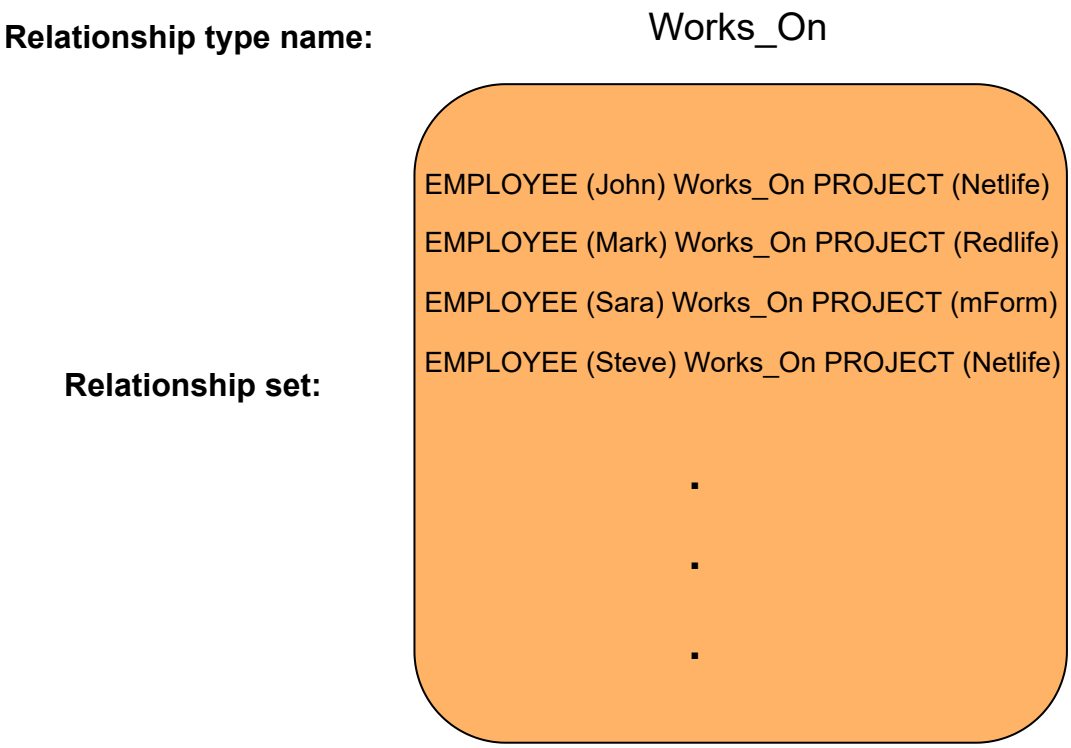
WE'LL COVER THE FOLLOWING

Relationships, relationship sets and relationship types

A relationship is an association between two entities, for example, an entity EMPLOYEE WORKS_ON PROJECT, which is another entity. However, note that different instances of employees will be working on different projects. So an employee, John WORKS_ON Project Netlife and maybe some other employee Sara (who is also an instance of an employee) WORKS_ON Project mForm.

So we can say that **relationship type** is simply the relationship that exists between two entities like WORKS_ON. While the set of similar associations at a point of time is called the **Relationship Set**.

The illustration below will help clear up the confusion:

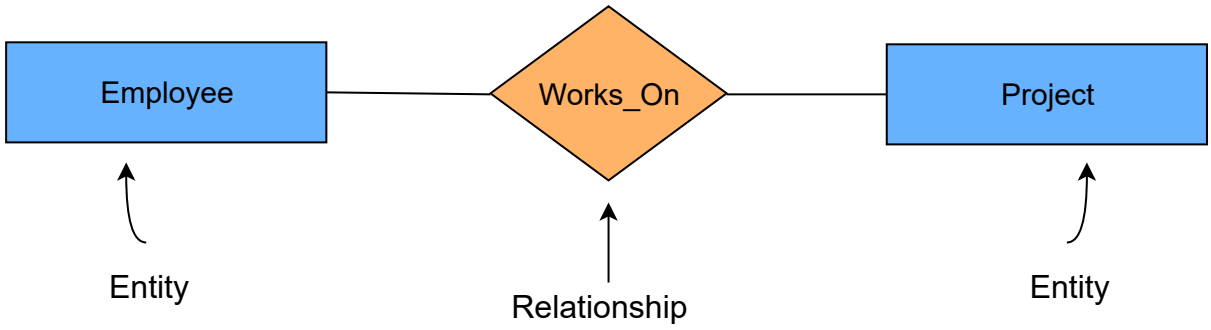


Example of relationship type and set

Similar to the case of entity types and entity sets, a relationship type and its corresponding relationship set are customarily referred to by the same name.

Representation in ER diagrams

In ER diagrams, relationship types are displayed as diamond-shaped boxes, which are connected by straight lines to the rectangular boxes that represent the participating entity types. The relationship name is displayed in the diamond-shaped box. The diagram below illustrates this concept:



Representation of relationships in ER model

In the next lesson, we will learn about the degrees of relationship type.

Degrees of Relationship Types

In this lesson, we will learn about the different degrees of relationship types.

WE'LL COVER THE FOLLOWING

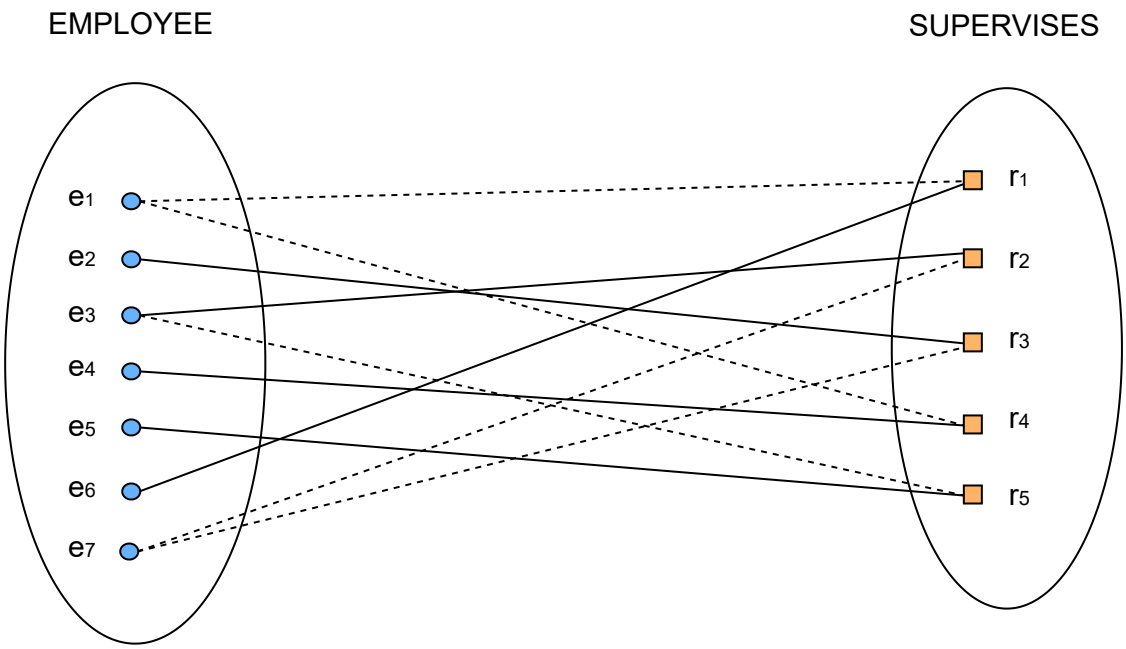
Degrees of relationship types

Consider the scenario where an employee works on multiple projects, so for a single EMPLOYEE entity the number of PROJECT entities he/she is associated with is multiple. Similarly, a PROJECT entity can have multiple EMPLOYEE entities that work on it. We represent this situation in our ER model through the degrees of a relationship.

The degree of a relationship type is the **number of participating entities types**. We will focus on mainly three types of degrees:

The Unary (recursive) relationship type

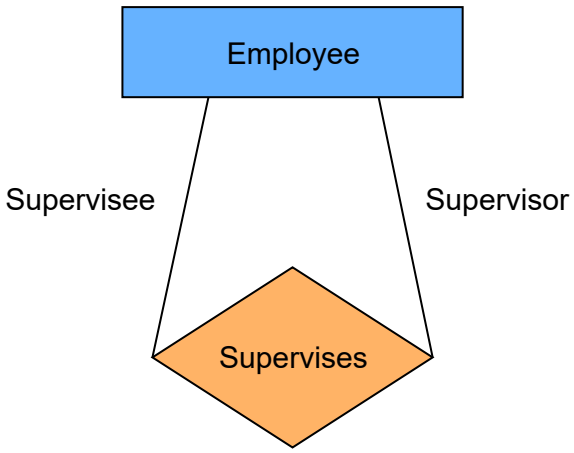
The unary relationship type involves only one entity type. However, the same entity type participates in the relationship type in different roles. For example, The SUPERVISES relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity set. Hence, the EMPLOYEE entity type participates twice in SUPERVISION: once in the role of supervisor, and once in the role of the supervisee. This concept is illustrated below:



Unary (Recursive) Relationship Type

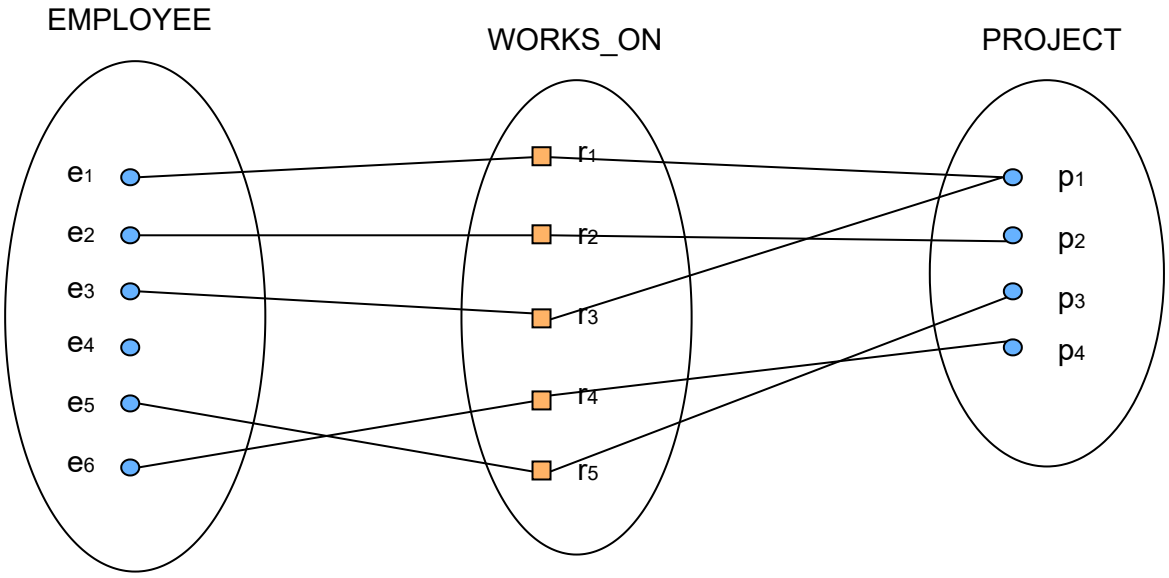
Each **relationship instance** r_i in SUPERVISES associates two different employee entities e_j and e_k, one of which plays the role of supervisor and the other the role of the supervisee. In the figure above, the dotted lines represent the supervisor role, and the solid lines represent the supervisee role; hence, e_1 supervises e_4 and e_6, e_7 supervises e_3 and e_2.

In the case of ER diagram we represent unary relationship types as:



The Binary relationship type

This relationship type has two entity types linked together. This is the most common relationship type. For example, consider a relationship type WORKS_ON between the two entity types EMPLOYEE and PROJECT, which associates each employee with the project he/she is working on. This relationship is expanded upon in the diagram below:



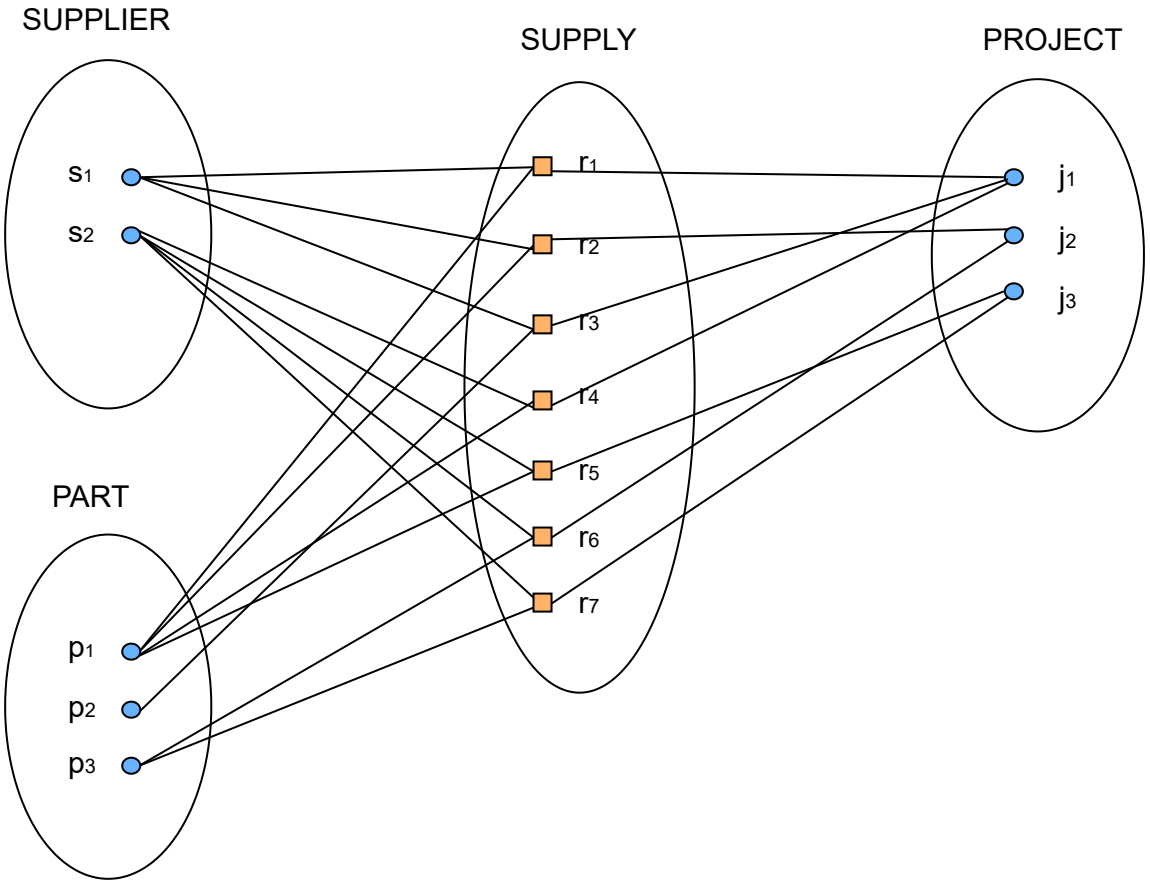
Binary Relationship Type

In the figure above each relationship instance r_i is shown connected to the EMPLOYEE and PROJECT entities that participate in r_i. In the mini-world represented by this figure, the employees e_1 and e_3 work on project p_1, the employee e_2 works on project p_2, and so on.

In the case of ER diagram we represent the binary relationship type as:

The Ternary relationship type

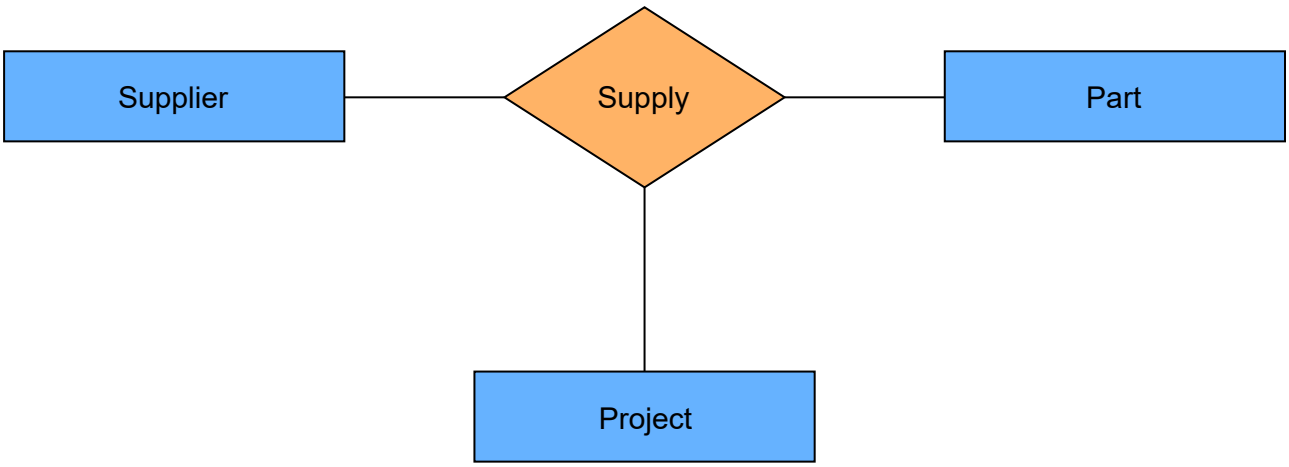
If there are three entity types linked together, the relationship is called a ternary relationship. An example of a ternary relationship is SUPPLY, shown in the figure below, where each relationship instance r_i associates three entities—a supplier s, a part p, and a project j.



Ternary Relationship Type

In the above diagram, we observe that a supplier s_i supplies part p_j to a project j_k. So we see that the supplier s_1 supplies part p_1 to both projects j_1 and j_2, while also supplying p_2 to project j_1. Similarly, other relationships can be determined as well.

In the case of an ER diagram we represent ternary relationship type as:



In the next lesson, we will discuss the constraints on binary relationship types.

Binary Relationship Type Constraints

In this lesson, we will look at the two major constraints on binary relationship types.

WE'LL COVER THE FOLLOWING

Binary relationship type constraints

In the previous lesson, we learned that there is a degree of relationship that exists between entities. However, sometimes this degree is affected by the constraints of the organization or a particular scenario. Consider, for example, a case where the company has a rule that each employee must work for exactly one department. In this and similar cases, we would like to describe this constraint in our schema. Such rules are usually called the "constraints" on the relationship types that exist in our schema.

These constraints limit the possible combinations of entities that may participate in the corresponding relationship set. There are two main types of binary relationship constraints: **mapping cardinality** and **participation**.

Let's look at each one of them in detail below.

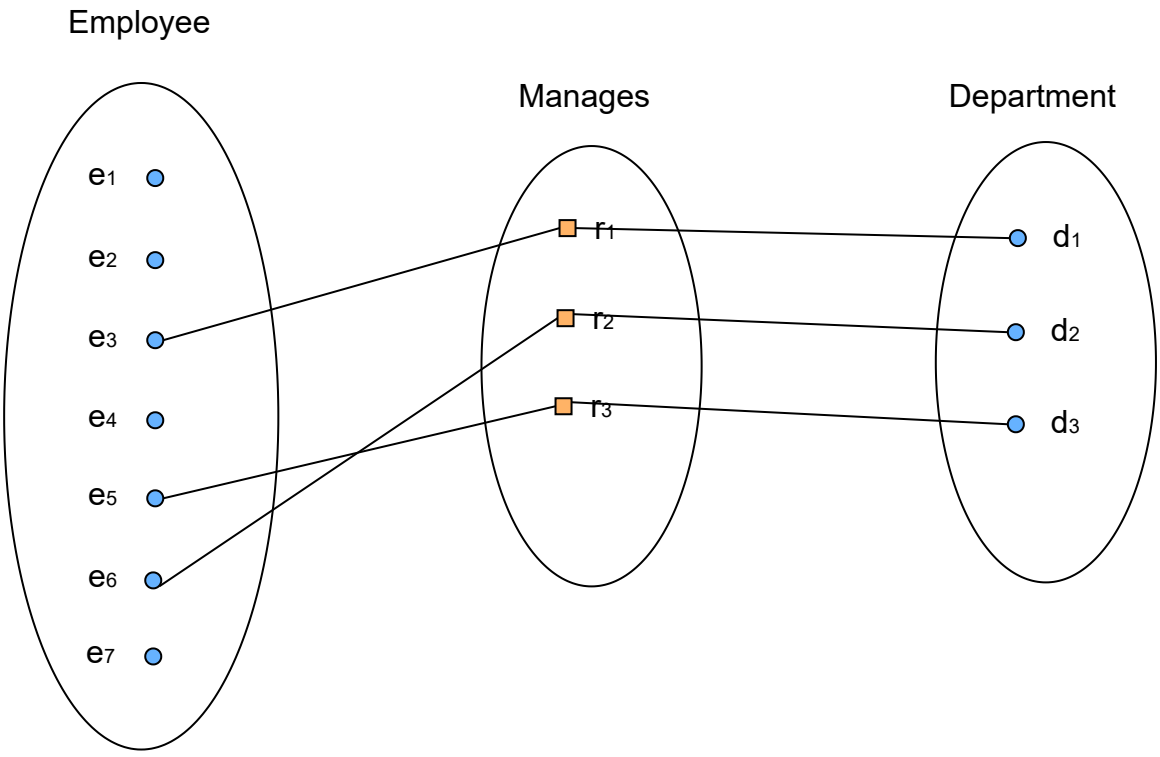
Mapping cardinality

Mapping cardinality describes the maximum number of entities that a given entity can be associated with via a relationship. In this lesson, we consider only the cardinality constraint for the binary relationship. The possible cardinality for binary relationship types are One to One (1:1), One to Many (1:N), and Many to Many (M:N).

The one to one relationship

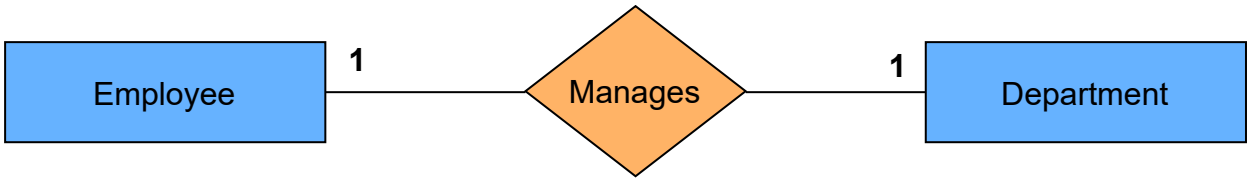
Given two entity sets A and B, there is a one to one relationship between A and B if each entity in set A is associated with at most one entity in set B and vice versa.

An example of a 1:1 binary relationship is MANAGES, which relates a DEPARTMENT entity to the EMPLOYEE entity. This represents the mini-world constraints that at any point in time, an employee can manage at most one department and a department can have at most one manager. This is represented in the diagram below:



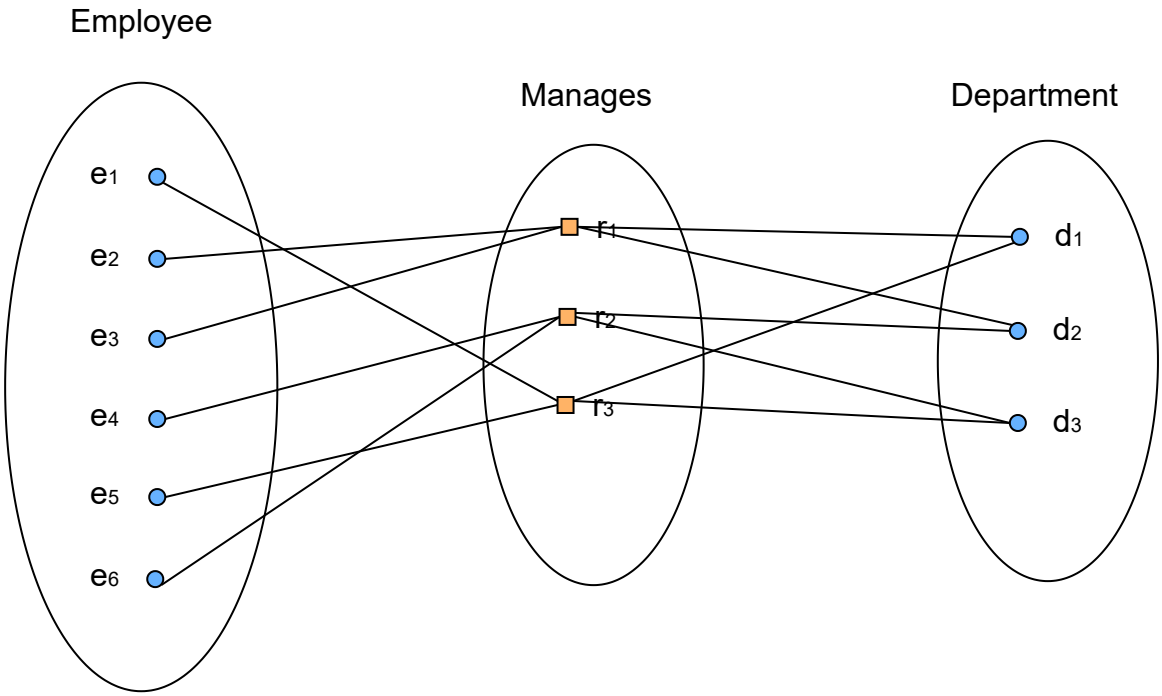
An example of a 1:1 relationship

Cardinality for binary relationships is represented on ER diagrams by displaying 1, M, and N on the diamonds. For a one to one relationship, it will look like this:



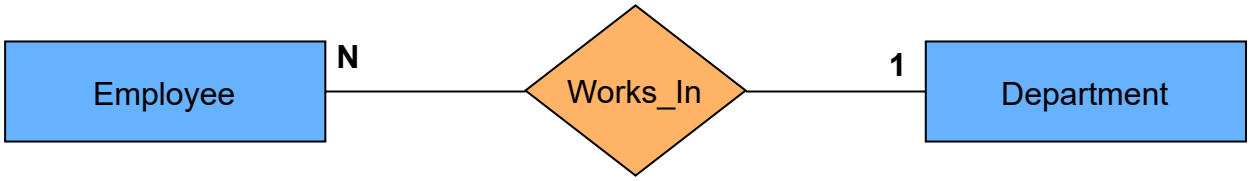
The one to many relationship

A one to many relationship set associates two entity sets A and B if each entity in A is associated with several entities in B however, each entity in B is associated with at most one entity in A. An example is that there are many employees working in a department, however, an employee can work for only one department. This can be seen below:



An example of a 1:N relationship

In the case of ER diagrams a 1:N relationship looks like this:

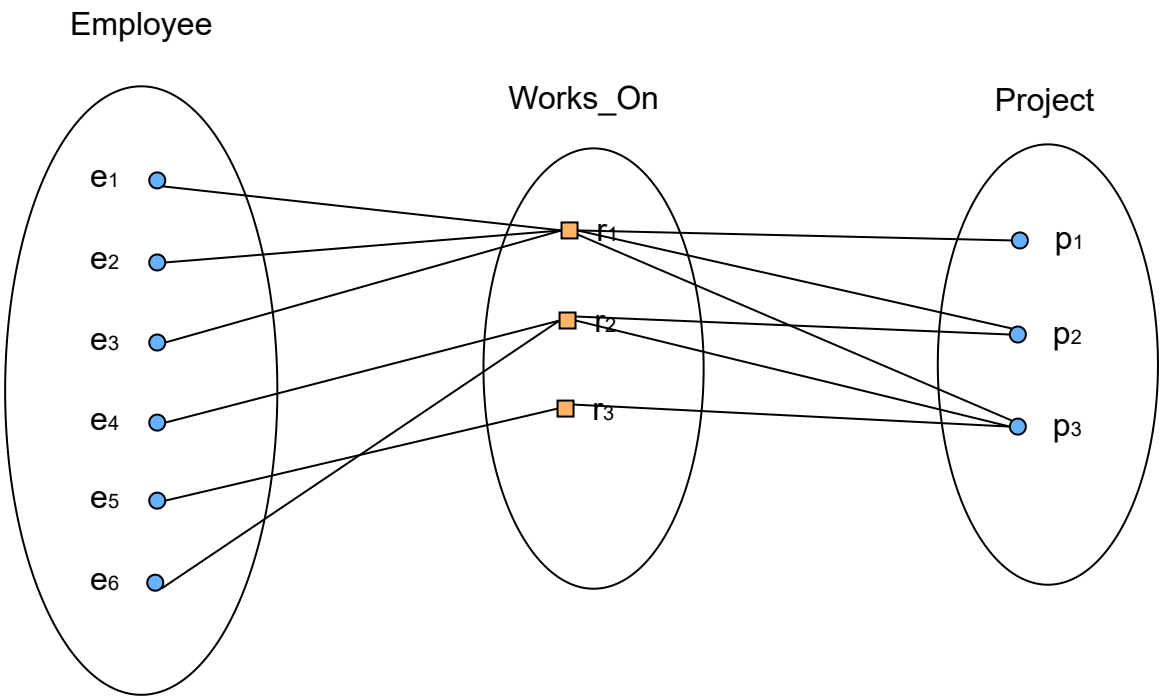


The many to many relationship

A many to many relationship set associates two entity sets A and B if each entity in A is associated with several entities in B, and, each entity in B is associated with several entities in A. The relationship type WORKS_ON is of

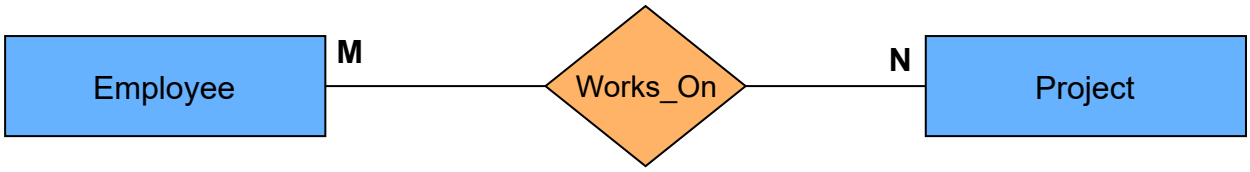
cardinality ratio M:N, because the mini-world rule is that an employee can

work on several projects and a project can have several employees. The diagram below can clear up any confusion:



An example of a M:N relationship

In the case of ER diagrams, an M:N relationship looks like this:



Participation

The participation constraint specifies whether the existence of an entity depends on it being related to another entity via the relationship type.

Participation in a relationship set R by an entity set A, maybe.

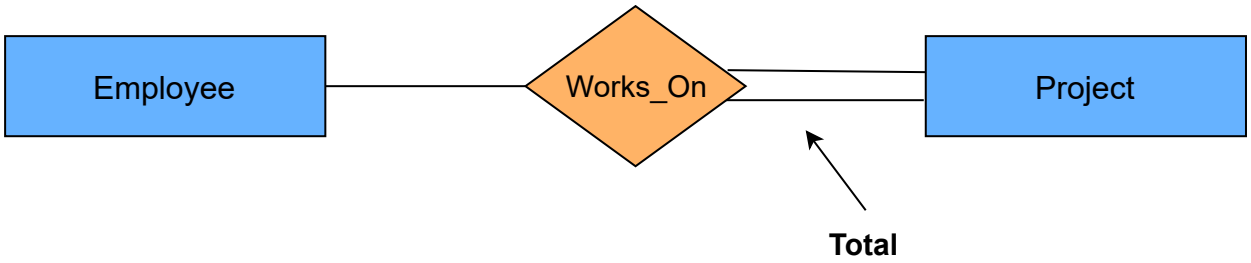
There are two types of participation constraints:

Total participation:

This specifies that each entity in the entity set must compulsorily participate in at least one relationship instance in that relationship set. That is why it is also known as **mandatory participation**. Total participation is represented using a double line between the entity set and relationship set.

using a double line between the entity set and relationship set.

Example: #

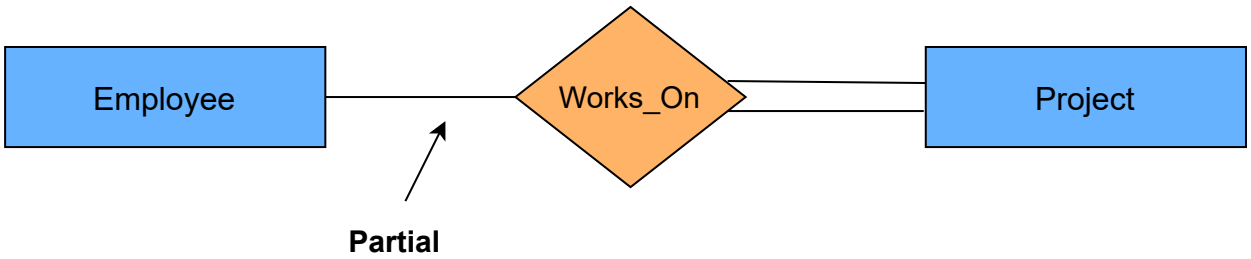


The double line between the PROJECT entity and relationship WORKS_ON signifies total participation. It specifies that each project must be assigned to at least one employee. In other words, without an EMPLOYEE entity, the PROJECT entity would not exist.

Partial participation:

This specifies that each entity in the entity set may or may not participate in the relationship instance in that relationship set. That is why it is also known as **optional participation**. Partial participation is represented using a single line between an entity set and a relationship set.

Example: #



A single line between the entity EMPLOYEE and the relationship WORKS_ON signifies partial participation. It specifies that some employees may work on a project and some may not.

In the next lesson, we will take a look at relationships with attributes.

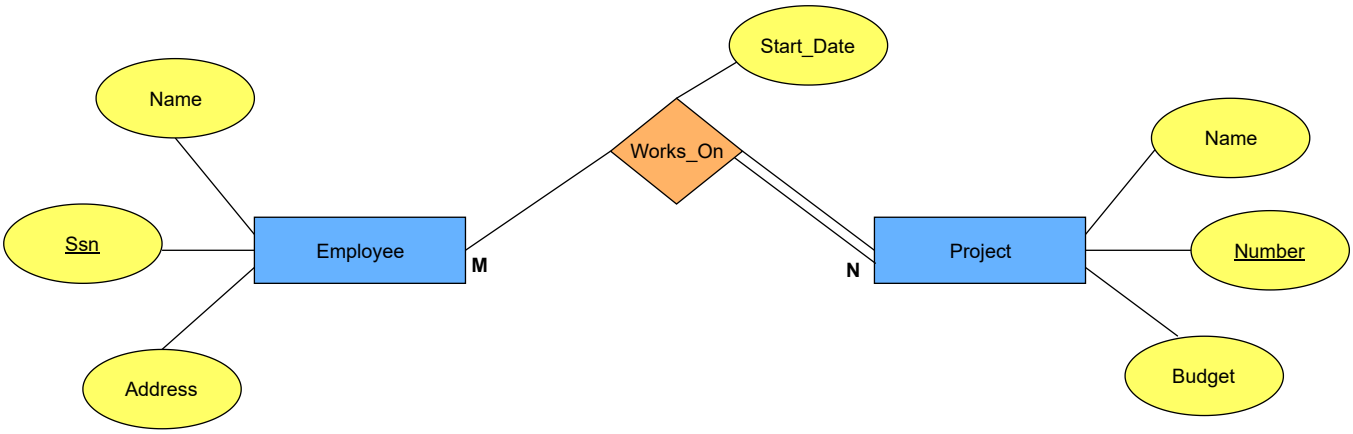
Attributes of Relationship Types

In this lesson, we will take a look at relationships with attributes.

WE'LL COVER THE FOLLOWING

Attributes of relationship types

Relationship types can also have attributes, similar to those of entity types. Hence the relationship WORKS_ON between EMPLOYEE and PROJECT has an attribute `Start_Date`. This is illustrated in the diagram below:



An example of an attribute associated with relationship types

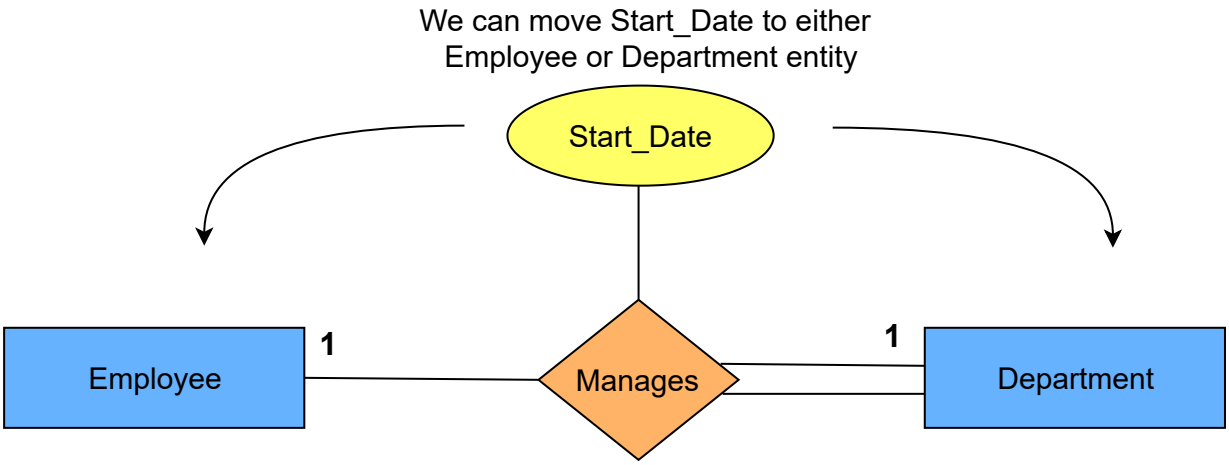
Generally, it is not recommended to give attributes to relationships if it's not required because when we represent this ER diagram in the database, we don't want to create a separate table for each relationship with attributes as this will create complexity.

So we want to move the attributes from the relationship to either of the two

entities joined by the relationship. Let's tackle this problem case by case with the help of examples:

The one to one relationship

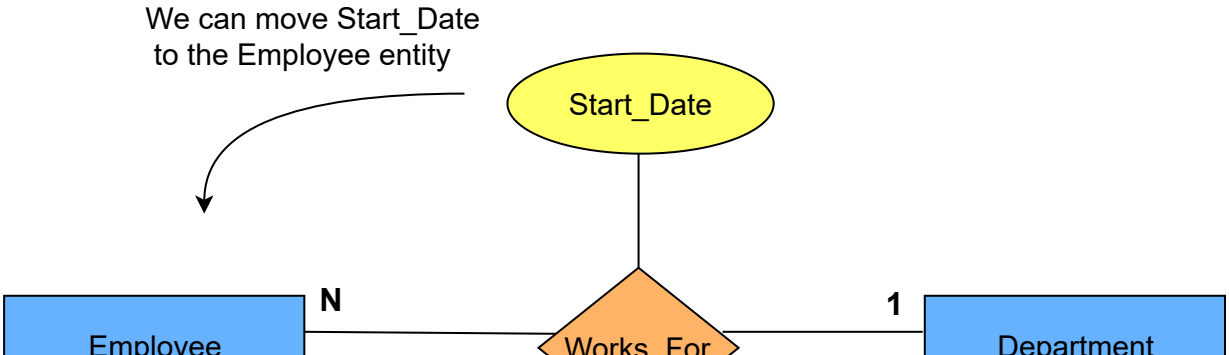
In the COMPANY database, an employee manages a department and each department is managed by an employee. Now, if we want to store the `Start_Date` from which the employee started managing the department then we may think that we can give the `Start_Date` attribute to the relationship MANAGES. But, in this case, we may avoid it by associating the `Start_Date` attribute to either the EMPLOYEE or DEPARTMENT entity.



The one to many relationship

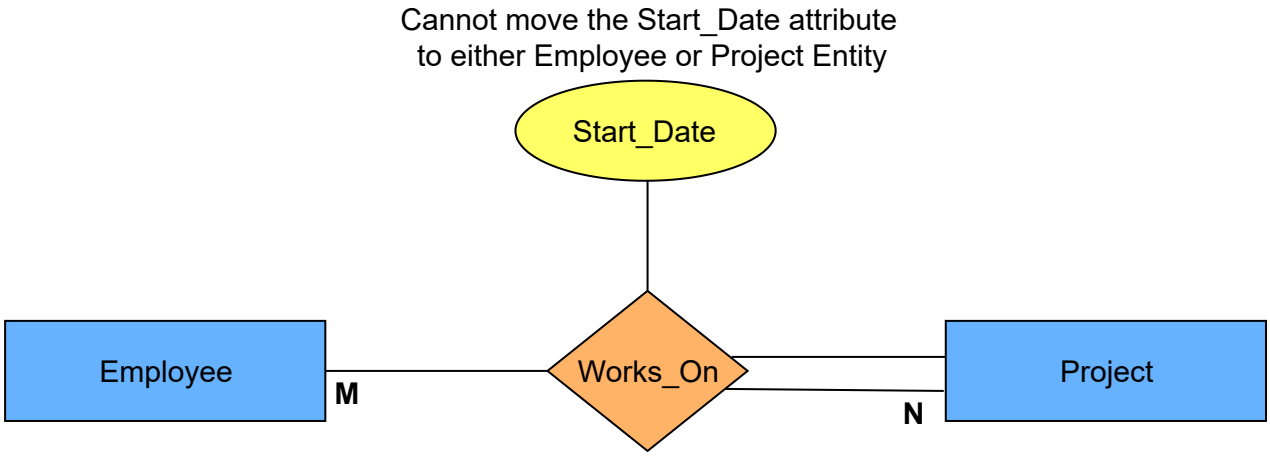
In our Company database, many employees can work for a department but each employee can work only for a single department. So, there is a one to many relationship between these entities.

Now if we want to store the `Start_Date` when the employee started working for the department, then instead of assigning it to the relationship we should assign it to the EMPLOYEE entity. Assigning it to the EMPLOYEE entity makes sense as each employee can work for a single department only, but on the other hand, one department can have many employees. Hence, it wouldn't make sense if we assign the `Start_Date` attribute to the DEPARTMENT entity.



The many to many relationship

In our Company database, an employee can work on many projects simultaneously and each project can have many employees working on it. Hence, it's a many to many relationship. So here assigning the `Start_Date` to the employee will not work as we will not know when an employee starts working on a specific project because a single employee can work on multiple projects each with its own starting date. It is the same case with the PROJECT entity. Hence, we are forced to assign the `Start_Date` attribute to the relationship WORKS_ON.



In the next lesson, we will learn about the last component in an ER model.

Weak Entity Types

In this lesson, we will look at the last component of our ER model.

WE'LL COVER THE FOLLOWING

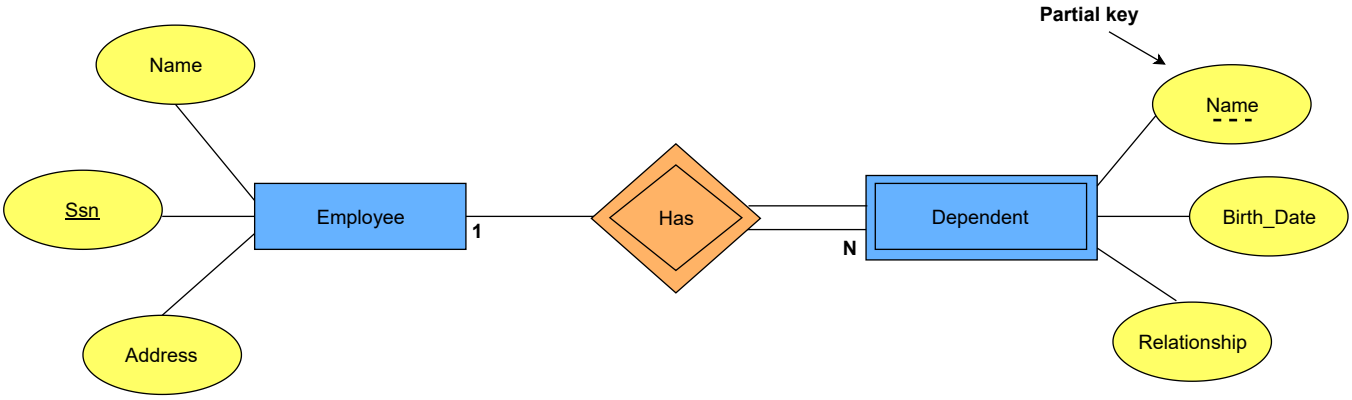
Weak entity types

Entity types that do not have key attributes of their own are called weak entity types like the DEPENDENT entity type in the company database.

In contrast, regular entity types that do have a key attribute are called **strong entity types**. Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with one of their attribute values. We call this other entity type the **identifying or owner entity type**, and we call the relationship type that relates a weak entity type to its owner the **identifying relationship** of the weak entity type.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship. In the diagram below, the attributes of DEPENDENT are `Name`, `Birth_Date`, and `Relationship` (to the employee). Two dependents of two distinct employees may, by chance, have the same values for `Name`, `Birth_Date`, and `Relationship`, but they are still distinct entities. They are identified as distinct entities only after determining the particular EMPLOYEE entity to which each dependent is related.

In ER diagrams, both a weak entity type and its identifying relationship are distinguished by surrounding their boxes and diamonds with double lines.



An example of a weak entity type

We also notice that a weak entity type normally has a **partial key**, which is the attribute that can uniquely identify weak entities that are related to the same owner entity. In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute `Name` of DEPENDENT is the partial key. In the ER diagram above, the partial key attribute is underlined with a dashed or dotted line.

Furthermore, a weak entity type always has a total participation constraint (existence dependency) with respect to its identifying relationship because a weak entity cannot be identified without an owner entity.

The next lesson will include an exercise where you will design an ER diagram for a university database.

Exercise 1

In this exercise, you will design the ER diagram for the UNIVERSITY database.

WE'LL COVER THE FOLLOWING

The university database

The university database stores details about university students, courses, the semester a student took a particular course (and his mark and grade if he completed it), and what degree program each student is enrolled in. The database is a long way from one that would be suitable for a large tertiary institution, but it does illustrate relationships that are interesting to visualize.

Consider the following requirements list:

Entities:

- STUDENT entity.
- PROGRAM entity.
- COURSE entity.

Attributes:

- Students have one or more given names, a surname, a student identifier, a date of birth, and the year they first enrolled. We can treat all given names as a single object—for example, “John Paul”
- A program has a name, a program identifier, the total credit points required to graduate, and the year it commenced.
- A course has a name, a course identifier, a credit point value, a year (for example, year 1), and a semester (for example, semester 1).

example, year 1), and a semester (for example, semester 1).

Relationships:

- The university offers one or more programs.
- A program is made up of one or more courses.
- A student must enroll in a program.
- A student takes the courses that are part of his/her program.
- When a student takes a course, the year and semester he/she attempted it are recorded. When he/she finishes the course, a grade (such as A or B) and a mark (such as 60 percent) are recorded.

Try to design the ER diagram on your own. Give it some time and if you have trouble remembering some of the concepts, feel free to take a look at the previous lessons.

In the next lesson, we will discuss the solution to this problem.

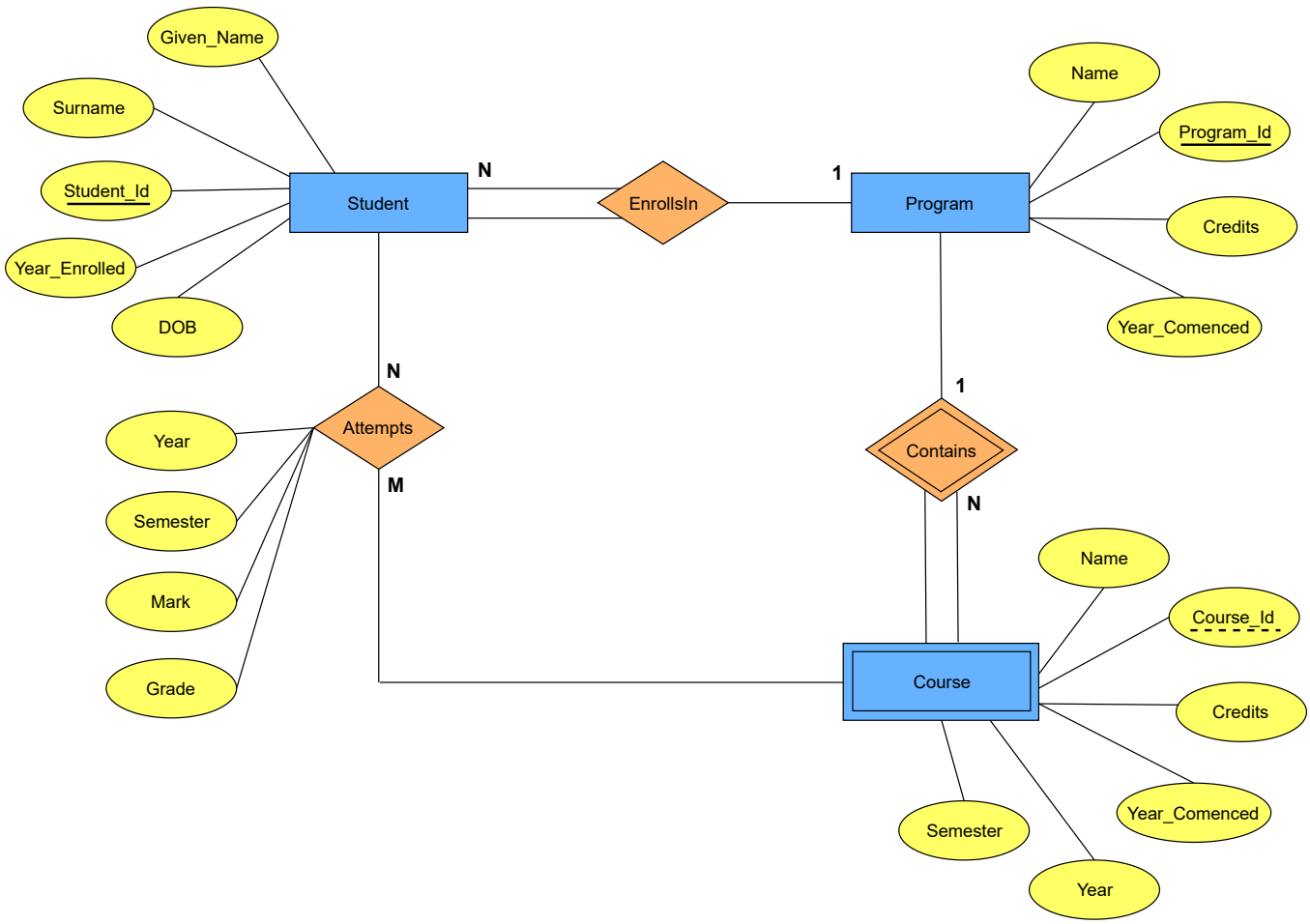
Solution to Exercise 1

Solution to exercise 1.

WE'LL COVER THE FOLLOWING

Solution

The ER diagram derived from our requirements is shown below. The diagram uses some advanced features, including relationships that have attributes and weak entity types.



Explanation

In our design:

- STUDENT is a strong entity, with an identifier, `Student_Id`, created to be the primary key used to distinguish between students (remember, we could have several students with the same name).
- PROGRAM is a strong entity, with the identifier `Program_Id` as the primary key used to distinguish between programs.
- Each student must be enrolled in a program, so the STUDENT entity participates totally in the many-to-one ENROLLS_IN relationship with PROGRAM. A program can exist without having any enrolled students, so it participates partially in this relationship.
- A COURSE has meaning only in the context of a PROGRAM, so it's a weak entity, with `Course_Id` as a weak key. This means that a COURSE entity is uniquely identified using its `Course_Id` and the `Program_Id` of its owning program.
- As a weak entity, COURSE participates totally in the many-to-one identifying relationship with its owning PROGRAM.
- STUDENT and COURSE are related through the many-to-many, ATTEMPTS relationships; a course can exist without a student, and a student can be enrolled without attempting any courses, so the participation is not total.
- When a student attempts a course, there are attributes needed to capture the `Year`, `Semester`, `Mark` and `Grade` of that course.

Exercise 2

In this exercise, you will design the ER model for flight database.

WE'LL COVER THE FOLLOWING

The flight database

The flight database stores details about an airline's fleet, flights, and seat bookings. Again, it's a hugely simplified version of what a real airline would use, but the principles are the same.

Consider the following requirements list:

Entities:

- AIRPLANE entity
- FLIGHT entity
- PASSENGER entity
- BOOKING entity

Attributes:

- An airplane has a model number, a unique registration number, and the capacity to take one or more passengers.
- An airplane flight has a unique flight number, a departure airport, a destination airport, departure date and time, and arrival date and time.
- A passenger has a first name, surname, and unique email address.

Relationships:

- Each flight is carried out by a single airplane.
- The airline has one or more airplanes.
- A passenger can book a seat on a flight.

Try to design the ER diagram on your own. Give it some time and if you have trouble remembering some of the concepts, feel free to take a look at the previous lessons.

In the next lesson, we will discuss the solution to this problem.

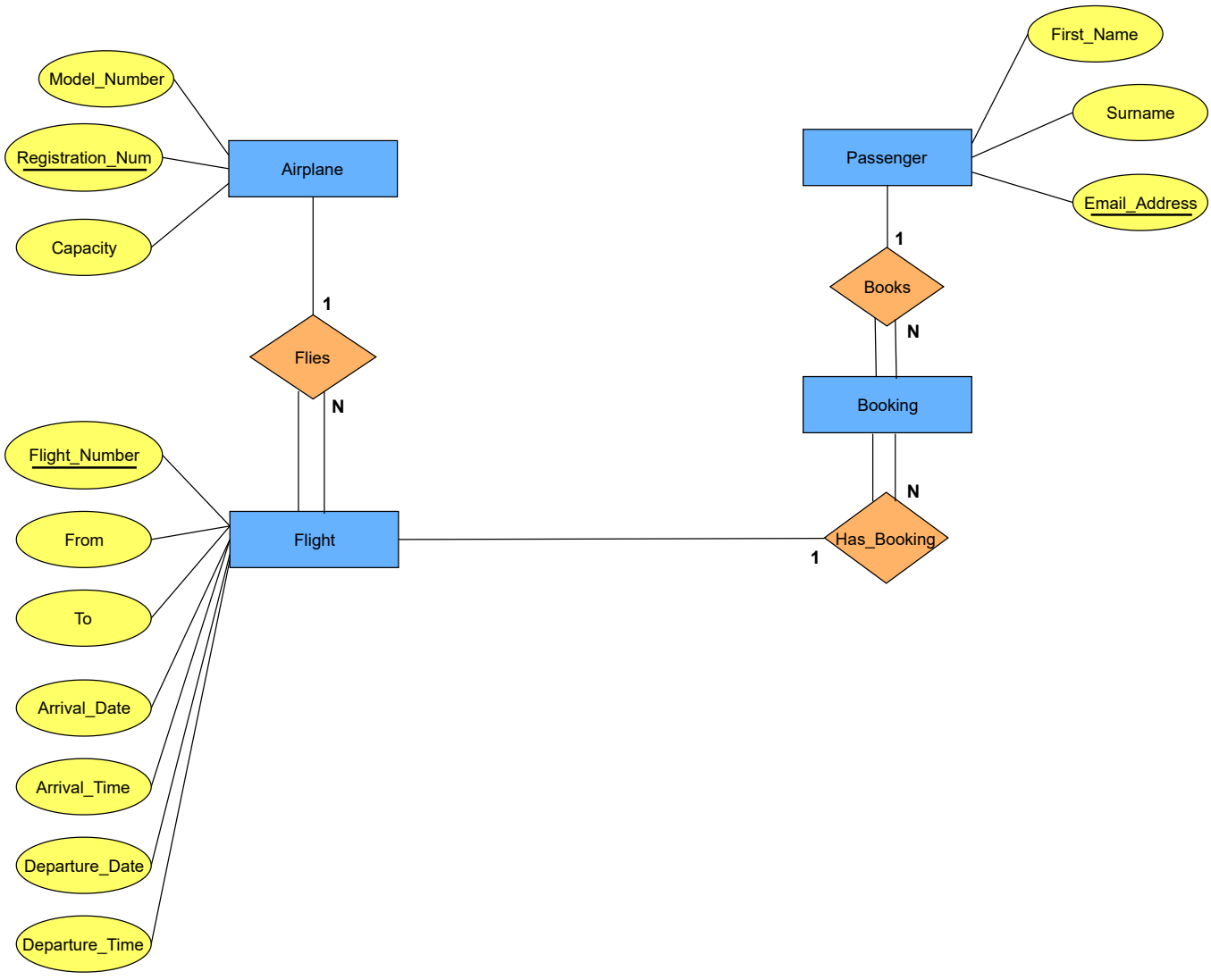
Solution to Exercise 2

Solution to exercise 2.

WE'LL COVER THE FOLLOWING

Solution

The ER diagram derived from our requirements is shown below.



Explanation

Explanation

In our design:

- An AIRPLANE is uniquely identified by its `Registration_Num`, so we use this as the primary key.
- A FLIGHT is uniquely identified by its `Flight_Number`, so we use the flight number as the primary key. The departure and destination airports are captured in the `From` and `To` attributes, and we have separate attributes for the departure and arrival date and time.
- Because no two passengers will share an email address, we can use the `Email_Address` as the primary key for the PASSENGER entity.
- An airplane can be involved in any number of flights, while each flight uses exactly one airplane, so the FLIES relationship between the AIRPLANE and FLIGHT entities has cardinality 1:N; because a flight cannot exist without an airplane, the FLIGHT entity participates totally in this relationship.
- A passenger can book any number of flights, while a flight can be booked by any number of passengers. We capture this by creating the entity BOOKING which has 1:N relationships between it and the PASSENGER and FLIGHT entities.

With this, the chapter comes to an end. By now we should be familiar with the different components that make up an ER diagram. And how to put them together to make an ER diagram.

In the next chapter, we will tackle the different concepts surrounding the relational database model.

Relational Model Concepts

In this lesson, we will discuss the fundamentals of relational databases.

WE'LL COVER THE FOLLOWING

Why use relational data models?

The relational data model was introduced by C.F. Codd in 1970. Currently, it is the most widely used data model.

The relational model has provided the basis for:

- Research on the theory of data/relationship/constraint.
- Numerous database design methodologies.
- The standard database access language called Structured Query Language (SQL).
- Almost all modern commercial database management systems.

The relational data model describes the world as “a collection of inter-related relations (or tables).”

What is relational model?

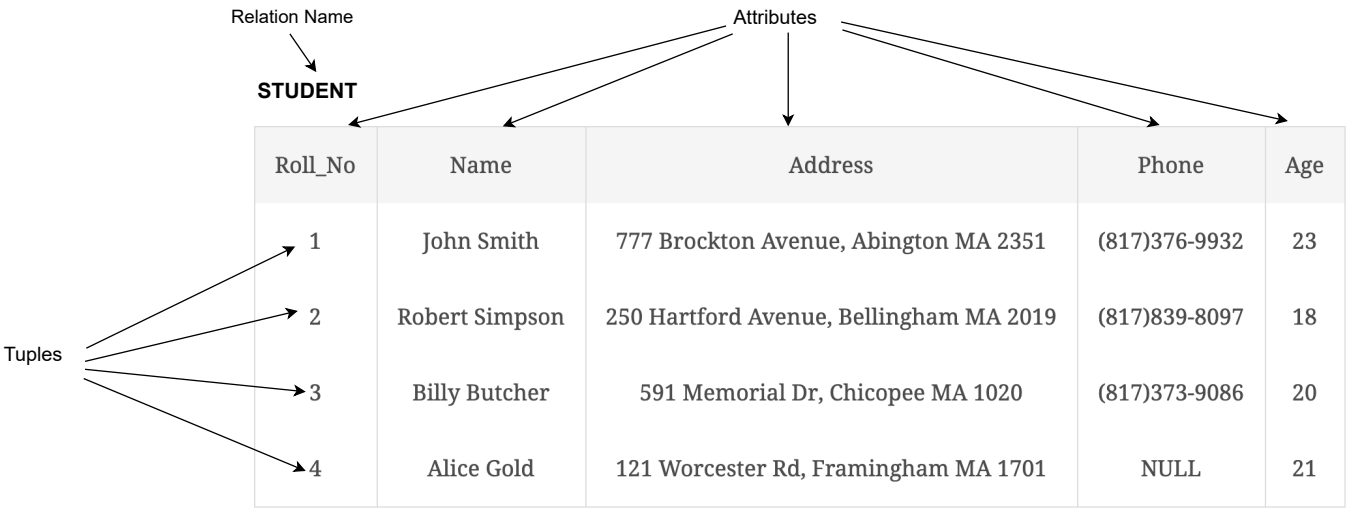
The relational model represents the database as a collection of relations. A **relation** is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship. The table and column names are helpful to interpret the meaning of values in each row.

Some popular relational database management systems are:

- DB2 and Informix Dynamic Server - IBM
- Oracle and RDB – Oracle
- SQL Server and Access - Microsoft

Fundamental concepts of the relational data model

Consider a relation STUDENT as shown below:



Roll_No	Name	Address	Phone	Age
1	John Smith	777 Brockton Avenue, Abington MA 2351	(817)376-9932	23
2	Robert Simpson	250 Hartford Avenue, Bellingham MA 2019	(817)839-8097	18
3	Billy Butcher	591 Memorial Dr, Chicopee MA 1020	(817)373-9086	20
4	Alice Gold	121 Worcester Rd, Framingham MA 1701	NULL	21

The STUDENT relation

1. **Attribute**: Each column in a table. Attributes are the properties that define a relation, e.g., `Roll_No`, `Name`, etc.
2. **Tuple**: It is nothing but a single row of a table, which contains a single record. The above relation contains 4 tuples.
3. **Relation schema**: A relation schema represents the name of the relation with its attributes. e.g; STUDENT (`Roll_No`, `Name`, `Address`, `Phone`, and `Age`) is a relation schema for the STUDENT relation.
4. **Degree**: The number of attributes in the relation is known as the degree of the relation. The STUDENT relation defined above has a degree of 5.
5. **Cardinality**: The number of tuples in a relation is known as cardinality. The STUDENT relation defined above has a cardinality of 4.
6. **Relation instance**: The set of tuples in a relation at a particular instance in time is called a relation instance. It changes whenever we insert, delete

in time is called a relation instance. It changes whenever we insert, delete or update the database.

7. **NULL values**: The value which is not known or unavailable is called a NULL value. In the STUDENT relation, we see that the phone number of a STUDENT whose `Roll_No` is 4 is set to NULL.

8. **Domain**: A domain is the original set of atomic values used to model data. By **atomic value**, we mean that each value in the domain is indivisible as far as the relational model is concerned. In other words, a domain is a set of acceptable values that a column is allowed to contain. For example:

    - Name: The set of character strings that represent the name of a person.

    - Age: Possible ages of students in a university; each must be an integer value between 17 and 27.

Now that you are familiar with the terminology used in relational models, in the next lesson we will highlight some of the important properties of relational tables.

Properties of a Table

In this lesson, we will outline some of the characteristics of relations (tables).

WE'LL COVER THE FOLLOWING

Properties of relational tables

Although we defined relations as a table of values, certain characteristics make a relation different from a table. We highlight these properties with the help of the following table:

Student Relation

Stud_Id	First_Name	Last_Name	Class	Major
1	Adam	Smith	Junior	CS
2	Jonathan	Joestar	Sophomore	Economics
3	Lucas	Klein	Senior	Physics
4	Brandon	Jones	Freshman	Biology

5	Ian	Damiani	Junior	Sociology

1. Each row is unique

This property ensures that no two rows in a relational table are identical; there is at least one column, or set of columns, whose values uniquely identify each row in the table. Such columns are called primary keys and are discussed in more detail in the next lesson.

In the STUDENT relation above, we can see that the `Stud_Id` attribute is unique for every student, so it can be used to identify each student tuple in the STUDENT relation.

2. Values are atomic

An **atomic value** is one that can not be broken down into smaller pieces. In other words, the table does not contain repeating groups or multivalued attributes.

The key benefit of the atomic value property is that it simplifies data manipulation logic.

As we can see, each column in our STUDENT relation contains simple data that cannot be broken down any further.

3. Column values are of the same kind

In relational terms, this means that all values in a column come from the same domain based on their data type including:

- number (numeric, integer, float, smallint,...)
- character (string)
- date
- logical (true or false)

This property simplifies data access because developers and users can be certain of the type of data contained in a given column. It also simplifies data validation.

This property is evident in our STUDENT relation as all records in the `Stud_Id` attribute are of the same type i.e. $integer$. While the rest of the attributes are of type $string$.

of type *string*.

4. The sequence of columns is insignificant

This property states that there is no specific sequence of columns, i.e., columns can be retrieved in any order and various sequences. The benefit of this property is that it enables many users to share the same relational table without concern of how the table is organized. It also permits the physical structure of the database to change without affecting the relations.

So, according to this property, even if we change the order of the columns, it will not have any impact on the workings of the relational model.

5. The sequence of rows is insignificant

This property is analogous to the one above but applies to rows instead of columns. The main benefit is that the rows of a relational table can be retrieved in any order or sequence. Adding information to a relational table is simplified and does not affect existing queries.

Similar to the above property, even if the rows are ordered differently, it will not affect the way the relational model works.

6. Each column has a unique name

Because the sequence of columns is insignificant, columns must be referenced by name and not by position. In general, a column name need not be unique within an entire database but only within the relation to which it belongs.

Since we have unique names for each of the columns in our STUDENT relation, this property is being satisfied.

In the next lesson, we will learn about the different types of keys that are present in relational databases.

Introduction to Database Keys

In this lesson, we will highlight some of keys that are present in databases.

WE'LL COVER THE FOLLOWING

Keys are a very important part of the relational database model. They are used to establish and identify relationships between tables and also to uniquely identify any record or row of data inside a table.

A key can be a single attribute or a group of attributes, where the combination may act as a key.

Why do we need a key?

In real-world applications, the number of tables required for storing data is huge, and different tables are related to each other as well.

Also, tables store a lot of data. Tables generally extend to thousands of records, unsorted and unorganized.

Now, to fetch any particular record from such a dataset, you will have to apply some conditions. But what if there is duplicate data present and every time you try to fetch some data by applying certain conditions, you get the wrong data? How many trials before you get the right data?

To avoid all this, keys are defined to easily identify any row of data in a table.

Let's try to understand what keys are all about using the example of a STUDENT table:

Std_Id	Name	Phone
1	John	(201)6723452
2	Adam	(202)1165674
3	Bruce	(480)7867898
4	James	(516)0080080

Super key

A super key is defined as a set of attributes within a table that can uniquely identify each record within a table.

In the table defined above, the super key would include `Std_Id`, (`Std_Id`, `Name`), `Phone` etc.

Confused? The first one is pretty simple as `Std_Id` is unique for every row of data, hence it can be used to identify each row uniquely.

Next comes, (`Std_Id`, `Name`), now `Name` of two students can be the same, but their `Std_Id` can't be same hence this combination can also be a super key.

Similarly, the phone number for every student will be unique (provided it is his/her mobile number), hence, again, `Phone` can also be a key.

So they all are super keys.

Candidate key

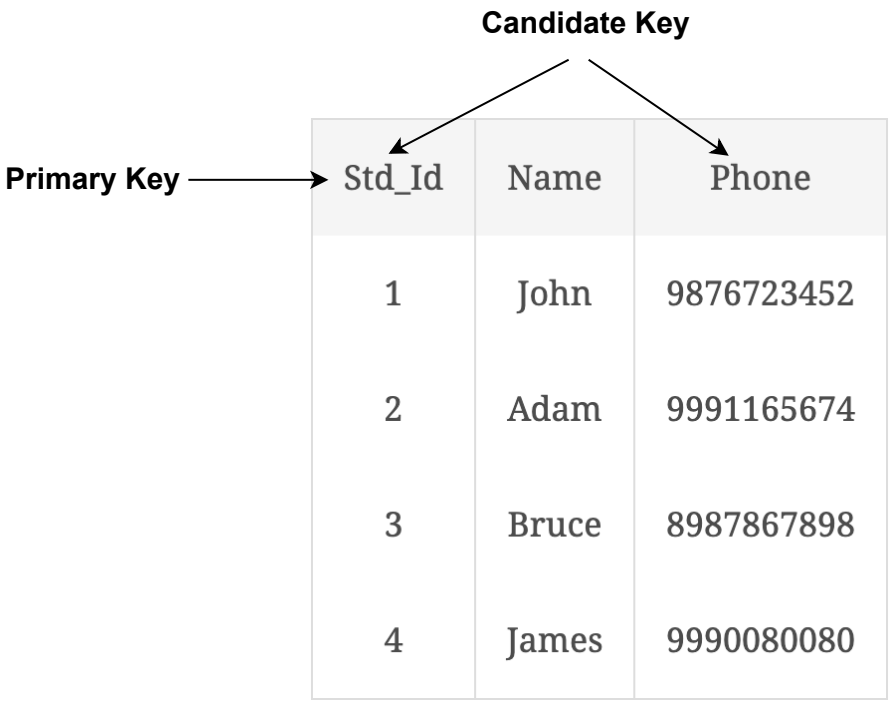
Candidate keys are defined as the minimal set of fields that can uniquely identify each record in a table. There can be more than one candidate key.

In our example, `Std_Id` and `Phone` both are candidate keys for the STUDENT table. The (`Std_Id`, `Name`) super key is not a candidate key as we can remove the `Name` field and still be able to uniquely identify each record.

Furthermore, a candidate key can never be `NULL` or empty. Its value should be unique. Also, a candidate key can be a combination of more than one column (attributes).

Primary key

There can be more than one candidate key in a relation, out of which one can be chosen as the primary key. For Example, `Std_Id` as well as `Phone` both candidate keys for relation STUDENT but `Std_Id` can be chosen as the primary key (only one out of many candidate keys).



Composite key

A key that consists of two or more attributes that uniquely identify any record in a table is called a composite key. But the attributes which together form the composite key are not a key independently or individually.

Std_Id	Subject_Id	Marks
1	8	70

1	12	90
2	12	65

In the above MARKS table, we have the marks scored by a student in a particular subject. In this table `Std_Id` and `Subject_Id` together will form the primary key, hence it is a composite key.

Alternate key

The candidate key other than the primary key is called an alternate key. For Example, `Std_Id` as well as `Phone`, are candidate keys for relation STUDENT but `Phone` will be the alternate key.

Foreign key

A foreign key is a column or group of columns in a relational database table that provides a link between the data in two tables. It acts as a cross-reference between tables because it references the primary key of another table, thereby establishing a link between them. Let's take a look at a simple example:

Student table:

Std_Id	Name	Phone
1	John	9876723452
2	Adam	9991165674
3	Bruce	8987867898
4	James	9990080080

Course table:

Std_Id	Course_No	Course_Name
1	C1	Software Engineering
2	C1	Software Engineering
1	C2	Networks

The `Std_Id` in the COURSE relation acts as the foreign key to `Std_Id` in the STUDENT relation.

It may be worth noting that unlike the primary key of any given relation, foreign keys can be NULL or may contain duplicate tuples, i.e., it need not follow uniqueness constraint.

For Example, `Std_Id` in the COURSE relation is not unique. It has been repeated for the first and third tuple. However, the `Std_Id` in STUDENT relation is a primary key and it needs to be always unique and it cannot be `NULL`.

In the next lesson, we will dive deep into the rules and constraints placed upon relational databases.

Integrity Rules and Constraints

In this lesson, we will discuss the different types of constraints present on relational databases.

WE'LL COVER THE FOLLOWING

Constraints are a very important feature in a relational model. In fact, the relational model supports the well-defined theory of constraints on attributes or tables. Constraints are useful because they allow a designer to specify the semantics of data in the database. Constraints are the rules that force DBMSs to check that data satisfies the semantics.

Relational integrity constraints

Relational integrity constraints are the conditions that must be present for a valid relation. These integrity constraints are derived from the rules in the mini-world that the database represents.

There are many types of integrity constraints, but we will focus on the following:

Domain constraints

Domain constraints specify that within each tuple, the value of each attribute must appear in the corresponding domain (in other words, it should belong to the appropriate data type). We have already discussed how domains can be specified in the previous lessons. The data types associated with domains typically include integers, real numbers, characters, booleans, etc. Let's

consider the following example:

Std_Id	Name	Age
1	Jack	22
2	Jill	25
3	Ahmed	19
4	Anderson	A

Here, value ‘A’ is not allowed since only integer values can be taken by the `Age` attribute.

Entity integrity

To ensure entity integrity, it is required that every relation has a primary key. Neither the primary key nor any part of it can contain `NULL` values. This is because the presence of a `NULL` value in the primary key violates the uniqueness property. Let’s say we have the following table with `Std_Id` as the primary key:

Std_Id	Name	Age
1	Jack	22
2	Jill	25
3	Ahmed	19
NULL	Anderson	21

This relation does not satisfy the entity integrity constraint as here the primary key contains a `NULL` value.

Referential integrity constraint

This constraint is enforced when a foreign key references the primary key of a relation. It specifies that all the values taken by the foreign key must either be available in the relation of the primary key or be `NULL`.

Referential integrity constraint has two very important results:

Rule 1: We cannot insert a record into a referencing relation if the corresponding record does not exist in the referenced relation.

Example: Let us consider two relations: STUDENT and DEPARTMENT. Here, STUDENT is the referencing relation while DEPARTMENT is the referenced relation as `Dep_No` acts as the foreign key in the STUDENT relation.

Std_Id	Name	Dep_No
1	Jack	D1
2	Jill	D4
3	Ahmed	D5

Dep_No	Dep_Name
D1	Physics
D4	Biology
D5	Computer Science
D7	Electrical Engineering

Now we insert the following tuple into the STUDENT table; <4, ‘Anderson’, ‘D9’>

Std_Id	Name	Dep_No
1	Jack	D1
2	Jill	D4
3	Ahmed	D5
4	Anderson	D9

We now see that the STUDENT relation does not satisfy the referential integrity constraint. This is because, in the DEPARTMENT relation, no value of a primary key specifies department no. 9. Hence, the referential integrity constraint is violated.

Rule 2: We cannot delete or update the record of the referenced relation if the corresponding record exists in the referencing relation.

Example: Again consider the STUDENT and DEPARTMENT relations as seen above.

Now if we try to delete the tuple in the DEPARTMENT relation with `Dep_No` = 'D1'. This will again violate the referential integrity constraint, as some students are already enrolled in the course and this will lead to inconsistencies in the database.

Key constraints

As we know, a primary key uniquely identifies each record in a table. Therefore, it must have a unique value for each tuple in the relation *R*. For example:

Std_Id	Name	Age
1	Jack	22
2	Jill	25

3	Ahmed	19
1	Anderson	21

The key constraint is violated as the primary key `Std_Id` has the same value for the first and fourth tuple.

In the next lesson, we will define what a relational database schema is and how it is used to represent integrity constraints.

Relational Database Schemas

In this lesson, we will discuss the basic concepts behind relational database schemas.

WE'LL COVER THE FOLLOWING

- Relational database schemas
- Representing referential integrity constraints in a schema

Relational database schemas

A relational database schema **S** is a set of relation schemas S = {R_1, R_2, ... , R_m} and a set of integrity constraints **IC**. A relational database state **DB** of S is a set of relation states DB = {r_1, r_2, ... , r_m} such that each r_i is a state of R_i. The figure below shows a relational database schema that we call COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, DEPENDENT}. In each relation schema, the underlined attribute represents the primary key.

EMPLOYEE

Name	Ssn	Bdate	Address	Salary	Super_Ssn	Dept_Num

Department

D_Name	D_No	Manager_Ssn

Dept_Locations

D_No	D_Location

Project

P_Name	P_Num	D_Num

Dependent

Essn	Dep_Name	Relationship

In the diagram above, the `D_No` attribute in both DEPARTMENT and DEPT_LOCATIONS stands for the same real-world concept—the number given to a department. That same concept is called `Dept_Num` in EMPLOYEE and `D_Num` in PROJECT. Attributes that represent the same real-world concept may or may not have identical names in different relations.

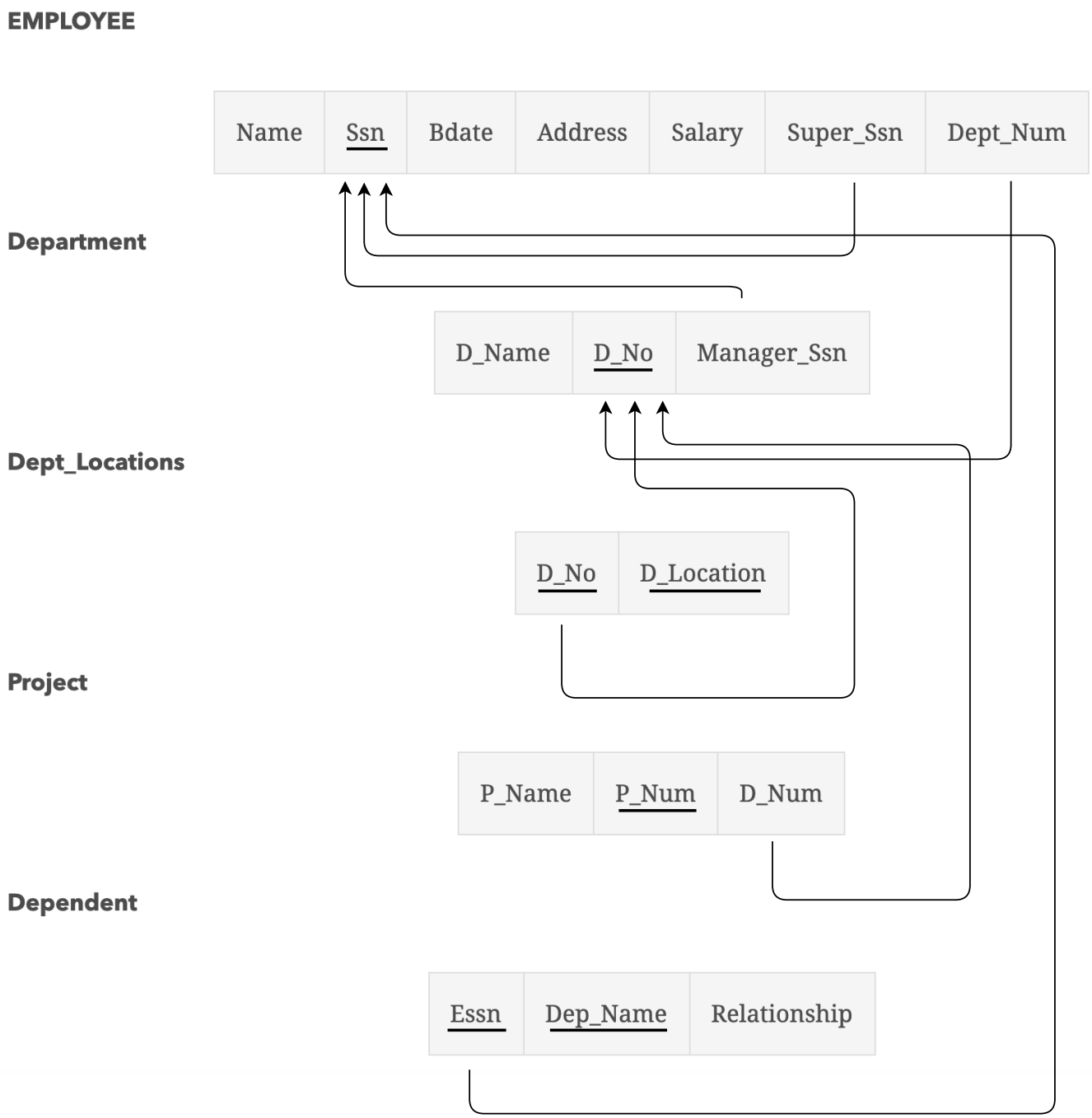
Representing referential integrity constraints in a schema

Referential integrity constraints typically arise from the relationships among the entities represented by the relation schemas. Consider the above example in which the attribute `Dept_Num` in the EMPLOYEE relation, refers to the department for which an employee works. Hence, we designate `Dept_Num` to

be a foreign key of EMPLOYEE referencing the DEPARTMENT relation.

This means that a value of `Dept_Num` in any tuple t_i of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT—the `D_No` attribute—in some tuple t_j of the DEPARTMENT relation. It could also be the case that the value of `Dept_Num` can be `NULL` if the employee does not belong to a department or will be assigned to a department later.

We can diagrammatically display referential integrity constraints by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation. The illustration below shows the schema in with the referential integrity constraints displayed in this manner:



Relational database schema of COMPANY database

From the above diagram, we can conclude that the `D_No` attribute in the DEPT_LOCATIONS table refers to the `D_NO` in the DEPARTMENTS table so we again draw an arrow to signify referential integrity constraint.

Also the `Manager_Ssn` attribute from the DEPENDENT table refers to the `Ssn` in the EMPLOYEE table. Since the manager is also an employee, the `Manager_Ssn` is derived from employee `Ssn`.

Similarly, the `Essn` attribute from the DEPENDENT table is a foreign key that refers to the `Ssn` in the EMPLOYEE table. If we need information regarding the parent of child (dependent) then we can use the `Essn` foreign key to retrieve that information from the EMPLOYEE table.

Furthermore, notice that a foreign key can refer to its own relation. For example, the attribute `Super_Ssn` in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, `Super_Ssn` is a foreign key that references the EMPLOYEE relation itself.

In the next lesson, we will discuss the different operations that can be carried out on relational databases.

Common Relational Database Operations

In this lesson, we will look at some of the different operations that can be performed on relational databases.

WE'LL COVER THE FOLLOWING

- The insert operation
- The delete operation
- The update operation

We will concentrate on the three basic operations that can change the states of relations in the database: Insert, Delete, and Update. **Insert** is used to insert one or more new tuples in a relation, **Delete** is used to delete tuples, and **Update** is used to change the values of some attributes in existing tuples.

Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated. So we will also discuss the types of constraints that may be violated by each of these operations along with the types of actions that may be taken if an operation causes a violation.

We will be using the database state illustrated below:

EMPLOYEE

Name	Ssn	Bdate	Salary	Super_Ssn	Dept_Num
John Smith	333445555	1968-05-22	45,000	NULL	3
Emily Taylor	987654321	1972-09-01	30,000	333445555	3

Adam Kovac	66688444 4	1969-04-09	55,000	33344555 5	3
Kevin Jaimes	888665555	1979-09-22	20,000	203948506	2

DEPARTMENT

D_Name	D_No	Manager_Ssn
Administration	3	333445555
Research	2	678884823

The insert operation

The insert operation provides a list of attribute values for a new tuple *t* that is to be inserted into a relation *R*.

The insert operation can violate any of the four types of constraints:

- Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type.
- Key constraints can be violated if the primary key value in the new tuple *t* already exists in another tuple in the relation.
- Entity integrity can be violated if any part of the primary key of the new tuple *t* is `NULL`.
- Referential integrity can be violated if the value of any foreign key in *t* refers to a tuple that does not exist in the referenced relation.

The following slides include examples of the insert operation:

Operation:
Insert <'James Willems', '334599005', '1970-01-29' , 75000, '333445555', 3> into EMPLOYEE

Result: This insertion is successful. So the table will be updated as follows:

Name	Ssn	Bdate	Salary	Super_Ssn	Dept_Num
John Smith	333445555	1968-05-22	45,000	NULL	3
Emily Taylor	987654321	1972-09-01	30,000	333445555	3
Adam Kovac	666884444	1969-04-09	55,000	333445555	3
Kevin Jaimes	888665555	1979-09-22	20,000	203948506	2
James Willems	334599005	1970-01-29	75,000	333445555	3

Table is updated with the new record.

Operation:
Insert <â Jack Dylanâ , NULL, â 1960-06-15â , 18000, NULL, 2> into EMPLOYEE.

Result: This insertion violates the entity integrity constraint as there is a NULL value for the primary key Ssn. Thus, this insertion will be rejected.

Name	Ssn	Bdate	Salary	Super_Ssn	Dept_Num
John Smith	333445555	1968-05-22	45,000	NULL	3
Emily Taylor	987654321	1972-09-01	30,000	333445555	3
Adam Kovac	666884444	1969-04-09	55,000	333445555	3
Kevin Jaimes	888665555	1979-09-22	20,000	203948506	2

Table remains the same.

Operation:
Insert <Bruce Greenâ€™, 690678901â€™, 1960-10-02â€™, 30000, 987654321â€™, 5> into EMPLOYEE.

Result: This insertion violates the referential integrity constraint specified on Dept_Num in EMPLOYEE because no corresponding referenced tuple exists in DEPARTMENT with D_No equal to 5.
So, again insertion will be rejected.

Name	Ssn	Bdate	Salary	Super_Ssn	Dept_Num
John Smith	333445555	1968-05-22	45,000	NULL	3
Emily Taylor	987654321	1972-09-01	30,000	333445555	3
Adam Kovac	666884444	1969-04-09	55,000	333445555	3
Kevin Jaimes	888665555	1979-09-22	20,000	203948506	2

Table remains the same.

3 of 3

If an insertion violates one or more constraints, the default option is to **reject** the insertion. In this case, it would be useful if the DBMS could provide a reason as to why the insertion was rejected.

The delete operation

The delete operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database. To specify deletion, a condition of the attributes of the relation selects the tuple (or tuples) to be deleted.

Here are a few examples of the delete operation:

Operation:
Delete the EMPLOYEE tuple with `Ssn` = '666884444'.

Result: This deletion is valid and the table will be updated as follows:

Name	Ssn	Bdate	Salary	Super_Ssn	Dept_Num
John Smith	333445555	1968-05-22	45,000	NULL	3
Emily Taylor	987654321	1972-09-01	30,000	333445555	3
Kevin Jaimes	888665555	1979-09-22	20,000	203948506	2

Table is updated with the deleted record.

2. Operation:
Delete the DEPARTMENT tuple with D_No = 3.

Result: This deletion is not acceptable, because there are tuples in EMPLOYEE relation that refer to this tuple via the Dept_Num foreign key. Hence, if this tuple is deleted, referential integrity will be violated.

D_Name	D_No	Manager_Ssn
Administration	3	333445555
Research	2	678884823

The DEPARTMENT table remains the same.

Several options are available if a deletion operation causes a violation. The first option, called **restrict**, is to reject the deletion. The second option, called **cascade**, is to attempt to cascade the deletion by deleting tuples that reference the tuple that is being deleted.

The update operation

The update operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation *R*. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified.

Here are a few examples of the update operation:

Operation:
Update the `Salary` of the EMPLOYEE tuple with Ssn = â□□888665555â□□ to 45

Result: this update operation is acceptable, and the table will change as follows:

Name	Ssn	Bdate	Salary	Super_Ssn	Dept_Num
John Smith	333445555	1968-05-22	45,000	NULL	3
Emily Taylor	987654321	1972-09-01	30,000	333445555	3
Adam Kovac	666884444	1969-04-09	55,000	333445555	3
Kevin Jaimes	888665555	1979-09-22	45,000	203948506	2

The appropriate record has been updated.

1 of 3

Operation:
Update the Dept_Num of the EMPLOYEE tuple with Ssn = â888665555â to 5.

Result: Unacceptable, because it violates referential integrity.

Name	Ssn	Bdate	Salary	Super_Ssn	Dept_Num
John Smith	333445555	1968-05-22	45,000	NULL	3
Emily Taylor	987654321	1972-09-01	30,000	333445555	3
Adam Kovac	666884444	1969-04-09	55,000	333445555	3
Kevin Jaimes	888665555	1979-09-22	20,000	203948506	2

The table remains the same.

Operation:
Update the Ssn of the EMPLOYEE tuple with Ssn = â888665555â to â98765

Result: Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple.

Name	Ssn	Bdate	Salary	Super_Ssn	Dept_Num
John Smith	333445555	1968-05-22	45,000	NULL	3
Emily Taylor	987654321	1972-09-01	30,000	333445555	3
Adam Kovac	666884444	1969-04-09	55,000	333445555	3
Kevin Jaimes	888665555	1979-09-22	20,000	203948506	2

The table remains the same.

Updating an attribute that is neither part of a primary key nor part of a foreign key usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain.

Modifying a primary key value is similar to deleting one tuple and inserting another in its place, because we use the primary key to identify tuples. Hence, the issues discussed earlier in insertion and deletion operations come into play.

The next lesson includes a quiz to test your knowledge regarding relational databases.

Quiz!

This quiz will test your knowledge of relational databases.

1 Which of the following is NOT a property of tables?

COMPLETED 0%

1 of 7

Congratulations! You have reached the end of this chapter. At this point, you should be comfortable with ideas such as keys, constraints, and operations regarding relational databases.

In the next chapter, we will tackle the concepts behind functional dependencies.

Intro to Functional Dependencies

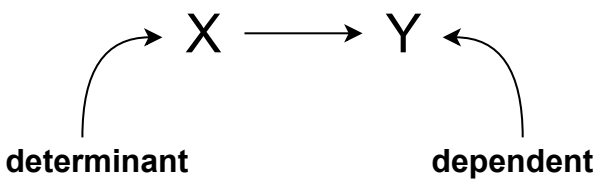
In this lesson, we will discuss the basic concepts of functional dependencies.

WE'LL COVER THE FOLLOWING

Introduction

One important theory developed for the entity relational (ER) model involves the notion of functional dependency (FD). The aim of studying this is to improve your understanding of relationships among data and to gain enough formalism to assist with practical database design.

A functional dependency is a relationship between two attributes, typically the primary key (PK) and other non-key attributes within a table. For any relation R, attribute Y is functionally dependent on attribute X, if, for every valid instance of X, that value of X uniquely determines the value of Y. This relationship is indicated by the representation below :



The left side of the above FD diagram is called the **determinant**, and the right side is the **dependent**.

Example

STUDENT relation

Student_Id	Name	Age
1	Spencer	22
2	Tony	20
3	Mark	23
4	Elyse	21

From the table above, it is obvious that `Student_Id` is the primary key as it is unique for each student.

So, following the above definition for functional dependencies, we conclude that:

- `Student_Id` $\rightarrow$ `Name`,
- `Student_Id` $\rightarrow$ `Age`,

Since we can determine both the name and age of the student through his/her `Student_Id`.

In the next lesson, we will look at the rules of functional dependencies.

Rules of Functional Dependencies

In this lesson, we will take a look at the different rules of FDs.

WE'LL COVER THE FOLLOWING

Armstrong's axioms

Armstrong's axioms are a set of inference rules used to infer all the functional dependencies on a relational database. They were developed by William W. Armstrong in 1974.

We will denote a set of attributes by the letters X, Y, and Z. Also, we will represent the union of two sets of attributes X and Y by *XY* instead of the usual $X \cup Y$; this notation is rather standard in database theory when dealing with sets of attributes.

We will now highlight the three primary rules:

Axiom of reflexivity

This axiom says, if Y is a subset of X, then X determines Y.

> If $Y \subseteq X$ then , $X \rightarrow Y$

For example, `Address` is composed of more than one piece of information; i.e. `House_No`, `Street`, and `State`. So according to the axiom of reflexivity `Address` (X) $\rightarrow$ `House_No` (Y) as `House_No` $\subseteq$ `Address`.

Axiom of augmentation

The axiom of augmentation, also known as a **partial dependency**, says if X determines Y, then XZ determines YZ for any Z.

> If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z

The axiom of augmentation says that every non-key attribute must be fully dependent on the whole composite primary key. To get a better understanding, look at the example below:

Std_Id	Course_Id	Name	Address	Age	Grade	Date_Completed
1	CS-100	Bob	777 Brockton Avenue, Abington MA 2351	22	A-	2019-10-09
1	PHY-101	Bob	777 Brockton Avenue, Abington MA 2351	22	C-	2019-10-10
2	CS-100	Jack	30 Memorial	20	B+	2019-10-12

			Drive, Avon MA 2322			

In the table above, we can see that `Std_Id` and `Course_Id` form a composite key (as the combination of these two attributes can be used to identify each tuple uniquely). However, we can also observe that the `Name`, `Address`, and `Age` attributes are only dependent on the `Std_Id`, not on the whole `Std_Id` and `Course_Id` composite key.

This situation is not desirable because every non-key attribute has to be fully dependent on the PK not just part of it. In this situation, student information is only partially dependent on the PK (`Std_Id`) which violates the axiom of augmentation.

We will discuss the solution to this problem when we study normalization in the next chapter.

Axiom of transitivity

The axiom of transitivity, also known as a **transitive dependency**, says if X determines Y, and Y determines Z, then X must also determine Z.

> If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Let's consider the following example:

Std_Id	Std_Name	Address	Age	Prog_Id	Prog_Name
1	Bob	777 Brockton Avenue, Abington MA 2351	22	CS-200	Introduction to Programming

1	Bob	777 Brockton Avenue, Abington MA 2351	22	PHY-100	Modern Physics
2	Jack	30 Memorial Drive, Avon MA 2322	20	CS-300	Advanced Programming

The table above has information not directly related to the student; for instance, `Prog_Name` is not dependent on `Std_Id`; it's dependent on `Prog_Id`.

This situation is not desirable because a non-key attribute (`Prog_Name`) depends on another non-key attribute (`Prog_Id`), which results in transitive dependency.

Again we will discuss the solution to this problem when we study normalization in the next chapter.

In the next lesson, we will look at how to display these dependencies in diagrammatic form.

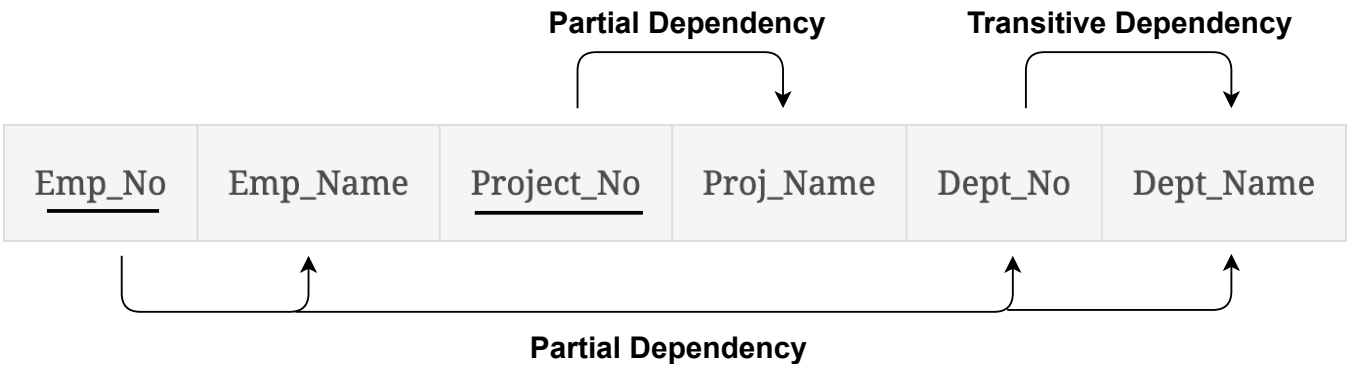
Dependency Diagrams

In this lesson, we will look at the diagrammatic representation of functional dependencies.

WE'LL COVER THE FOLLOWING

Dependency diagrams

A dependency diagram illustrates the various dependencies that might exist in a **non-normalized** table. A non-normalized table is one that has data redundancy in it. This is illustrated below:



An example of a dependency diagram

As we can observe from the table above, `Project_No` and `Emp_No`, combined, are the primary key (as the combination of these two attributes can be used to identify each record uniquely).

The following dependencies can be identified from this table:

Partial dependencies:

- `Project_No` → `Proj_Name`

Reason: Since the name of the project is only dependent upon part of the multi-attribute PK, i.e., Project_No, this results in partial dependency.

- `Emp_No` → `Emp_Name`, `Dept_No`

Reason: Similar to the example above, we see that these two attributes only depend on part of the composite key and not the whole.

Transitive dependency:

- `Dept_No` → `Dept_Name`

Reason: Since a non-key attribute (`Dept_Name`) is dependent on another non-key attribute (`Dept_No`), this results in a transitive dependency.

We have highlighted some of the dependencies that exist in this table, but there are many more.

The next lesson will include a short quiz to test your knowledge of functional dependencies.

Quiz!

Quiz on functional dependencies.

1

Which axiom states that every non-key attribute must be fully dependent on the whole composite primary key?

COMPLETED 0%

1 of 4

This brings us to the end of this chapter. By now you should be familiar with the rules and types of dependencies, as well as the representation of dependencies in our database schemas.

In the next chapter, we will see how functional dependencies are removed through normalization.

What Is Normalization?

In this lesson, we will learn about the different normalization forms and why we use the normalization technique.

WE'LL COVER THE FOLLOWING

Normalization should be part of the database design process. However, it is difficult to separate the normalization process from the ER modeling process so the two techniques should be used concurrently.

We use the entity relation diagram (ERD) to provide the big picture, or macro view, of an organization's data requirements and operations. This is created through an iterative process that involves identifying relevant entities, their attributes, and their relationships.

The normalization procedure focuses on the characteristics of specific entities and represents the micro view of entities within the entity-relationship diagram.

What is normalization?

Normalization is the branch of relational theory that provides design insights. It is the process of determining how much redundancy exists in a table. The goals of normalization are to:

- Be able to characterize the level of redundancy in a relational schema.
- Provide mechanisms for transforming schemas in order to remove redundancy.

Normalization draws heavily on the theory of functional dependencies.

Normalization theory defines six normal forms (NF). Each normal form involves a set of dependency properties that a schema must satisfy and each normal form gives guarantees about the presence and/or absence of update anomalies. This means that higher normal forms have less redundancy, and as a result, fewer update problems.

Normal forms

There are six normal forms, but we will only look at the first four, which are:

- First normal form (1NF)
- Second normal form (2NF)
- Third normal form (3NF)
- Boyce-Codd normal form (BCNF)

Practical uses of normal forms

Most practical design projects in commercial and governmental environments acquire existing designs of databases from previous designs, from designs in legacy models, etc. They are certainly interested in the fact that the designs are of good quality and are sustainable over long periods of time. Existing designs are evaluated by applying the tests for normal forms, and normalization is carried out in practice so that the resulting designs are of high quality and have minimal data redundancy.

In the next lesson, we will discuss the first normal form in detail.

First Normal Form

In this lesson, we will take a look at first normal form with some examples.

WE'LL COVER THE FOLLOWING

First normal form (1NF)

First normal form (1NF) states that the domain of an attribute must include only **atomic** (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a single tuple. The only attribute values permitted by 1NF are single atomic (or indivisible) values. In other words, if a relation contains a multi-valued attribute, it violates 1NF.

To make things more clear, let's consider the following examples.

Example 1

STUDENT Relation

Roll_No	Name	Course
101	Micheal	Databases, Operating Systems
102	Huber	Computer Networks, Data Structures

The STUDENT relation is not in 1NF because of the multi-valued attribute `Course`. To convert this table into 1NF, we must make sure that the values for the `Course` attribute are atomic. This can be seen below

Roll_No	Name	Course
101	Micheal	Databases
101	Micheal	Operating Systems
102	Huber	Computer Networks
102	Huber	Data Structures

Now this table is in 1NF.

Example 2

STUDENT Relation

Roll_No	Name	Phone
101	Felix	(123) 456–7890
102	Giorno	(456) 686–7821, (789) 316–9880
103	Tom	NULL

Again we can see that the second student has two phone numbers in a single tuple which violates 1NF. The table can be seen in the first normal form below:

Roll_No	Name	Phone
101	Felix	(123) 456–7890
102	Giorno	(456) 686–7821
102	Giorno	(789) 316–9880
103	Tom	NULL

In the next lesson, we will look at how the second normal form reduces data redundancy.

Second Normal Form

In this lesson, we will discuss how to decompose a table into second normal form and see some examples.

WE'LL COVER THE FOLLOWING

Second normal form (2NF)

To be in second normal form, a relation must be in first normal form (1NF) and it must not contain any partial dependencies. So a relation is in 2NF as long as it has no partial dependencies, i.e., no non-prime attributes (attributes which are not part of any candidate key) is dependent on any proper subset of a composite primary key of the table.

Example

STUDENT Relation

Stud_Id	Course_Id	Course_Fee
1	C1	1000
2	C2	1500
1	C4	2000
4	C3	1000
4	C1	1000

2	C5	3000

We can determine a few things by looking at the table above. First of all, `Course_Fee` alone cannot be used to identify each tuple uniquely. Furthermore, the combination of `Course_Fee` together with `Stud_Id` or `Course_Id` also cannot be used to uniquely identify each tuple. Hence, `Course_Fee` would be a non-prime attribute, as it does not belong to the composite primary key {`Stud_Id`, `Course_Id`}.

However, from the table, it is evident that `Course_Id` $\rightarrow$ `Course_Fee`, i.e., `Course_fee` is dependent on `Course_Id` only, which is a proper subset of the primary key. This results in a partial dependency and so this relation is not in 2NF.

To convert the above relation to 2NF, we need to split the table into two other tables such as:

- Table 1: `Stud_Id`, `Course_Id`
- Table 2: `Course_Id`, `Course_Fee`

Table 1

Stud_Id	Course_Id
1	C1
2	C2
1	C4
4	C3
4	C1
2	C5

Table 2

Course_Id	Course_Fee
C1	1000
C2	1500
C3	1000
C4	2000
C5	3000

In the first table, we keep `Course_Id` as the foreign key so that we can link the two tables together. This allows us to fetch the fee of a particular course from table 1.

It is important to note that 2NF tries to reduce the redundant data being stored in memory. For instance, if 100 students are taking the C1 course, we don't need to store its fee for all 100 records (tuples). Instead, we can store it in the second table just once.

In the next lesson, we will discuss the concepts behind the third normal form (3NF).

Third Normal Form

In this lesson, we will discuss the concept behind 3NF using an example.

WE'LL COVER THE FOLLOWING

Third normal form (3NF)

For a table to be in the third normal form:

1. It should be in the second normal form.
2. It should not have transitive dependency.

Example

SCORE Table

Std_Id	Subject_Id	Marks_obtained	Exam_Type	Total_Marks
1	CS-100	50	Final	100
2	CS-100	70	Final	100
3	CS-100	85	Final	100
1	Math-101	30	Mid-term	50
1	PHY-100	10	Practical	30

2	CHEM-100	20	Practical	30
3	PHY-120	40	Mid-term	50

From the table, we can see that the primary key for our SCORE table is a composite key, which means it's made up of two attributes (columns): {`Std_Id`, `subject_Id`}.

The column `Exam_Type` depends on both `Std_Id` and `Subject-Id`. For example, a student taking a chemistry course will have a practical lab exam but a student in a mathematics course will not. So we can say that `Exam_Type` is dependent on the whole composite key, thus there is no partial dependency, so the table is in 2NF.

But what about the column `Total_Marks`? Does it depend on our SCORE table's primary key?

Well, the column `Total_Marks` depends on `Exam_Type` since the type of exam the total score changes. For example, practicals are worth fewer marks while theory exams are worth more marks.

This results in a transitive dependency because a non-prime attribute depends on other non-prime attributes rather than depending upon the prime attributes or primary key.

So, in order to convert this table into 3NF, we take out the attributes `Exam_Type` and `Total_Marks` from the SCORE table and put them in their own table called the EXAM table. We will also add another column called `Exam_Id` in the EXAM table to act as the primary key. This column will also be added to the SCORE as a foreign key, so now we have a link between the two tables.

This is illustrated below:

SCORE table

Std_Id	Subject_Id	Marks_obtained	Exam_Id

EXAM table

Exam_Id	Exam_Type	Total_Marks

		allied	
1	CS-100	50	1
2	CS-100	70	1
3	CS-100	85	1
1	Math-101	30	2
1	PHY-100	10	3
2	CHEM-100	20	3
3	PHY-120	40	2

1	Final	100
2	Mid-term	50
3	Practical	30

In the next lesson, we will learn about our final normal form which is the Boyce-Codd normal form.

Boyce-Codd Normal Form

In this lesson, we will learn about Boyce-Codd normal form with the help of an example.

WE'LL COVER THE FOLLOWING

Boyce-Codd normal form (BCNF)

For a table to satisfy the Boyce-Codd normal form, it should satisfy the following two conditions:

1. It should be in the third normal form.
2. And, for any dependency A → B, A should be a super key.

The second point sounds a bit tricky, right? In simple words, it means that for a dependency A → B, A cannot be a non-prime attribute if B is a prime attribute.

Example

ENROLMENT table

Std_Id	Subject	Professor
101	CS-100	Eddie Jessup
101	MATH-101	Charles Kingsfield
102	CS-100	Robert Langdon

102	CHEM-101	John Keating
103	CS-100	Eddie Jessup

In the table above:

- One student can enroll in multiple subjects. For example, the student with `Std_Id` 101, has opted for two subjects: CS-100 and MATH-101.
- `Std_Id` and `Subject` together form the primary key as we can uniquely identify all the tuples in the table.
- One more important point to note here is that one professor teaches only one subject, but one subject may have two different professors. For example, CS-100 is taught by two different professors.

Hence, there is a dependency between the subject and professor where the subject depends on the professor's name.

This table satisfies the 1NF because all the values are atomic, column names are unique and all the values stored in a particular column are of the same domain.

This table also satisfies the 2NF as there is no partial dependency. There is also no transitive dependency, hence the table also satisfies the 3NF.

But this table is not in the Boyce-Codd normal form. The reason being that `Std_Id` and `Subject` form a composite primary key, which means `Subject` is a prime attribute.

But, there is one more dependency, `Professor` → `Subject`, and while `Subject` is a prime attribute, `Professor` is a non-prime attribute, which is not allowed by BCNF.

So, to make this relation (table) satisfy BCNF, we will decompose this table into two tables: the STUDENT table and the PROFESSOR table.

STUDENT table

PROFESSOR table

Std_Id	Professor_Id
101	1
101	2
102	3
102	4
103	1

Professor_Id	Professor	Subject
1	Eddie Jessup	CS-100
2	Charles Kingsfield	MATH-101
3	Robert Langdon	CS-100
4	John Keating	CHEM-101

In the PROFESSOR table, both `Professor` and `Subject` are non-prime attributes and they both depend on the primary key i.e. `Professor_Id`. So, these tables are in BCNF because a prime attribute is not dependent on a non-prime attribute. We also include `Professor_Id` in the STUDENT table as a foreign key so that the two tables can be linked together.

The next lesson will test your knowledge of second normal form.

Exercise 1

This exercise will test your knowledge of 2NF.

WE'LL COVER THE FOLLOWING

- Problem Statement

Problem Statement

First, determine whether the following table is in second normal form. If it is not in 2NF, then convert it into 2NF.

Customer-Order Table

Cust_Id	Cust_Name	Order_Id	Order_Detail	Order_Category
1	Jack	34	Shampoo	Hygiene
2	Bruce	22	TV	Electronics
3	Amanda	84	Shirts	Clothing
4	James	12	Shoes	Clothing
2	Bruce	62	Glasses	Clothing
5	Veronica	84	Shirts	Clothing

Take your time with this problem. Take a look at the second normal form lesson if you have trouble remembering the concept.

Hint: Start with identifying the primary key.

We will discuss the solution in the next lesson.

Solution to Exercise 1

In this lesson we will discuss the solution to exercise 1.

WE'LL COVER THE FOLLOWING

Solution

The second normal form states that it should meet all the rules for 1NF and there must be no partial dependencies between any of the columns with the primary key.

Customer-order table

Cust_Id	Cust_Name	Order_Id	Order_Detail	Order_Category
1	Jack	34	Shampoo	Hygiene
2	Bruce	22	TV	Electronics
3	Amanda	84	Shirts	Clothing
4	James	12	Shoes	Clothing
2	Bruce	62	Glasses	Clothing
5	Veronica	84	Shirts	Clothing

First, we can see that the table above is in the first normal form; it obeys all

the rules of the first normal form.

Secondly, the primary key consists of the `Cust_Id` and the `Order_Id`. Combined, they are unique assuming the same customer would not order the same thing.

However, the table is not in the second normal form because there are partial dependencies of primary keys and columns. `Cust_Name` is dependent on `Cust_Id` and there's no real link between a customer's name and what he/she purchased. The `Order_Deatil` and `Order_Category` are also dependent on the `Order_Id`, but they are not dependent on the `Cust_Id`, because there is no link between a `Cust_Id` and an `Order_Detail` or their `Order_Category`.

To make this table comply with the second normal form, you need to separate the columns into three tables.

First, create a table to store the customer details as shown below:

Customer table

Cust_Id	Cust_Name
1	Jack
2	Bruce
3	Amanda
4	James
5	Veronica

The next step is to create a table to store the details of each order:

Orders table

Order_Id	Order_Detail	Order_Category

34	Shampoo	Hygiene
22	TV	Electronics
84	Shirts	Clothing
12	Shoes	Clothing
62	Glasses	Clothing

Finally, create a third table storing just the `Cust_Id` and the `Order_Id` to keep track of all the orders for a customer:

Customer-order table

Cust_Id	Order_Id
1	34
2	22
3	84
4	12
2	62
5	84

The third table is simply used to link the first two tables.

Now all the tables are in 2NF as there is no partial dependency between any column.

Exercise 2

This exercise will test your knowledge of 3NF.

WE'LL COVER THE FOLLOWING

- Problem statement

Problem statement

First, determine whether the following table is in third normal form. If it is not in 3NF, then convert it into 3NF.

Customer table

Cust_Id	Cust_Name	DOB	Street	City	State	Zip
1	Jack	1996-01-13	777 Brockton Avenue	Abington	MA	2351
2	Bruce	1995-09-22	3018 East Ave	Central Square	NY	13036
3	Amy	1999-11-17	80 Town Line Rd	Rocky Hill	CT	6067
4	James	1998-	5710	Northp	AL	35476

4	James	1998-03-10	3710 Mcfarland Blvd	Northport	AL	35476
5	Veronica	1990-06-09	2900 Pepperrell Pkwy	Opelika	AL	36801

Take your time with this problem. Take a look at the third normal form lesson if you have trouble remembering the concept.

We will discuss the solution in the next lesson.

Solution to Exercise 2

In this lesson we will discuss the solution to exercise 2.

WE'LL COVER THE FOLLOWING

Solution

A table is in third normal form when the following conditions are met:

- It is in the second normal form.
- All non-primary fields are dependent on the primary key.

Customer table

Cust_Id	Cust_Name	DOB	Street	City	State	Zip
1	Jack	1996-01-13	777 Brockton Avenue	Abington	MA	2351
2	Bruce	1995-09-22	3018 East Ave	Central Square	NY	13036
3	Amy	1999-11-17	80 Town Line	Rocky Hill	CT	6067

			Rd			
4	James	1998-03-10	5710 Mcfarland Blvd	Northport	AL	35476
5	Veronica	1990-06-09	2900 Pepperrell Pkwy	Opelika	AL	36801

First, we can see that the table above is in the first normal form; it obeys all the rules of the first normal form.

Secondly, the primary key consists of the `Cust_Id` as it uniquely identifies each record in the table.

Therefore the table is in second normal form as there is no composite primary key.

However, the table is not in the third normal form because the street name, city, and state are unbreakably bound to their zip code. The dependency between the zip code and the address is called transitive dependency. To comply with the third normal form, all you need to do is to move the `Street`, `City`, and `State` fields into their own table.

Zip code table

Zip	Street	City	State
2351	777 Brockton Avenue	Abington	MA
13036	3018 East Ave	Central Square	NY
6067	80 Town Line Rd	Rocky Hill	CT

	Rd		
35476	5710 Mcfarland Blvd	Northport	AL
36801	2900 Pepperrell Pkwy	Opelika	AL

The next step is to alter the CUSTOMER table as shown below:

Customer table

Cust_Id	Cust_Name	DOB	Zip
1	Jack	1996-01-13	2351
2	Bruce	1995-09-22	13036
3	Amy	1999-11-17	6067
4	James	1998-03-10	35476
5	Veronica	1990-06-09	36801

The `Zip` field acts as a foreign key in the CUSTOMER table so that we can get the address details of the corresponding customer. It acts as a link between the two tables.

Now all the tables are in 3NF as there is no transitive dependency between any column.

Quiz!

This quiz will test your knowledge of normalization and the different normal forms.

1 What is/are the goals of normalization?

COMPLETED 0%

1 of 5

We have reached the end of this chapter, good job! By now we should be familiar with the concept of normal forms and how to normalize a non-normalized table.

In the next chapter, we will dive deep into the concept behind Structured Query Language (SQL).

Structured Query Language (SQL)

This lesson introduces users to SQL.

WE'LL COVER THE FOLLOWING

What is SQL?

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data in a relational database.

SQL is the standard language for a relational database system. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, and SQL Server use SQL as their standard database language.

A brief history of SQL

SQL was initially developed by IBM in the early 1970s. The initial version, called SEQUEL (Structured English Query Language), was designed to manipulate and retrieve data stored in IBM's quasi-relational database management system, System R. Then, in the late 1970s, Relational Software Inc., which is now Oracle Corporation, introduced the first commercially available implementation of SQL, Oracle V2 for VAX computers.

Why SQL?

SQL is widely popular because it can

- Create the database and table structures.

- Perform basic data management chores (add, delete and modify).
- Perform complex queries to transform raw data into useful information.

SQL Commands

The standard SQL commands to interact with relational databases are **CREATE**, **SELECT**, **INSERT**, **UPDATE**, **DELETE**, and **DROP**. These commands can be classified into the following groups based on their nature:

DDL - Data Definition Language

Command	Description
CREATE	Creates a new table, a view of a table, or other objects in the database.
ALTER	Modifies an existing database object, such as a table.
DROP	Deletes an entire table, a view of a table or other objects in the database.

DML - Data Manipulation Language

Command	Description
SELECT	Retrieves certain records from one or more tables.
INSERT	Creates a record.
UPDATE	Modifies records.

DELETE	Deletes records.

In the next lesson, we will highlight some important data types and operations in SQL.

SQL Data Types and Operators

In this lesson, we will learn about the different data types supported by SQL.

WE'LL COVER THE FOLLOWING

SQL data types

A SQL data type is an attribute that specifies the type of data of any object. You can specify the data type of each column in the table based on your requirements.

Some of them are listed below:

Exact Numeric Data Types

Data Type	Ranges From	To
int	-2,147,483,648	2,147,483,647
bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	-10^38 +1	10^38 -1
numeric	-10^38 +1	10^38 -1

Approximate Numeric Data Types

Data Type	Ranges From	To
float	-1.79E + 30	1.79E + 308
real	-3.40E + 38	3.40E + 38

Date and Time Data Types

Data Type	Ranges From	To
datetime	Jan 1, 1753	Dec 31, 9999
smalldatetime	Jan 1, 1900	Jun 6, 2079
date	Stores a date like June 30, 1991	-
time	Stores a time like 12:30 P.M.	
-		

Character Strings Data Types

DATA TYPE	Description
char	Maximum length of 8,000 characters.(Fixed length non-Unicode characters)
varchar	Maximum of 8,000 characters. (Variable-length non-Unicode data).
varchar(max)	Maximum length of 2E + 31 characters, Variable-length non-Unicode data.
text	Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters.

What is an operator in SQL?

An operator is a reserved word or character used primarily in a SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. These operators are used to specify conditions in a SQL statement and to serve as conjunctions for multiple conditions in a statement.

SQL arithmetic operators

Assume variable 'a' holds 10 and variable 'b' holds 20.

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	a + b will give 30
(Subtraction)	Subtracts right hand	a - b will give -10

- (Subtraction)	Subtracts right-hand operand from left-hand operand	a - b will give -10
* (Multiplication)	Multiplies values on either side of the operator	a * b will give 200
/ (Division)	Divides left-hand operand by right-hand operand	b / a will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder	b % a will give 0

SQL comparison operators

Assume ‘variable a’ holds 10 and ‘variable b’ holds 20.

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true	(a = b) is not true
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true	(a != b) is true

<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true	(a <> b) is true
>	Checks if the value of the left operand is greater than the value of right operand, if yes then condition becomes true	(a > b) is not true
<	Checks if the value of the left operand is less than the value of right operand, if yes then condition becomes true	(a < b) is true
>=	Checks if the value of the left operand is greater than or equal to the value of right operand, if yes then condition becomes true	(a >= b) is not true
<=	Checks if the value of the left operand is less than or equal to the value of right operand, if yes then condition becomes true	(a <= b) is true

You don’t have to worry about these terms too much as we will be using them in the lessons to come, which will help you to understand how they are used. In the next lesson, we will discuss SQL constraints.

SQL Constraints

In this lesson, we will take a look at the different constraints enforced on columns and tables.

WE'LL COVER THE FOLLOWING

SQL constraints

Constraints are the rules enforced on data columns in a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints can either be column level or table level. Column level constraints are applied only to one column, whereas table level constraints are applied to the entire table.

The following are some of the most commonly used constraints available in SQL:

The NOT NULL constraint

Ensures that a column cannot have a `NULL` value. By default, a column can hold `NULL` values. If you do not want a column to have a `NULL` value, then you need to define such a constraint on this column specifying that `NULL` is now not allowed for that column. A `NULL` is not the same as no data, rather, it represents unknown data.

The DEFAULT constraint

Provides a default value for a column when none is specified.

The UNIQUE constraint

Ensures that all the values in a column are different. The UNIQUE constraint prevents two records from having identical values in a column. In the CUSTOMER table, for example, you might want to prevent two or more people from having the same ID.

The PRIMARY key constraint

Uniquely identifies each row/record in a database table. A primary key is a field that uniquely identifies each row/record in a database table. A primary key must contain unique values and cannot have `NULL` values. A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a composite key.

The FOREIGN Key constraint

Uniquely identifies a row/record in any other database table. A foreign key is a key used to link two tables together. This is sometimes also called a referencing key. It can be a column or a combination of columns whose values match a primary key in a different table.

The CHECK constraint

Ensures that all values in a column satisfy certain conditions. It enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and isn't entered into the table. For example, in the CUSTOMER table, we can check if the customer is over 18 years old by applying the CHECK constraint on the `Age` attribute (column).

Get excited as we are going to write our first query in the next lesson.

CREATE, DROP, and USE Databases

In this lesson, we will look at the syntax for creating and deleting a database.

WE'LL COVER THE FOLLOWING

CREATE DATABASE

The SQL **CREATE DATABASE** statement is used to create a new SQL database.

Syntax

The basic syntax of this `CREATE DATABASE` statement is as follows:

```
CREATE DATABASE DatabaseName;
```

The database name should always be unique within the RDBMS.

Keep in mind that SQL keywords are NOT case sensitive: `create` is the same as `CREATE`.

Also, some database systems require a semicolon at the end of each SQL statement. A semicolon is the standard way to separate each SQL statement in

database systems that allow more than one SQL statement to be executed in the same call to the server.

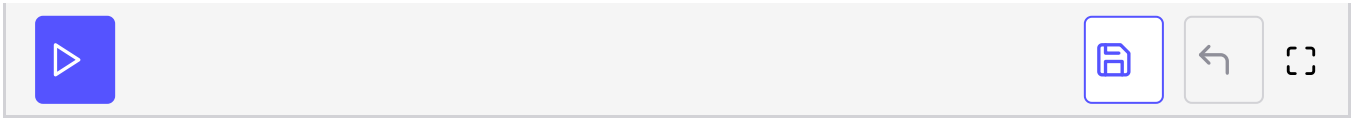
Example

If you want to create a new database, for example, testDB1, then the `CREATE DATABASE` statement would be as shown below:

```
CREATE DATABASE testDB1;
```

Now let's create two databases in the code below:

```
CREATE DATABASE testDB1;
CREATE DATABASE testDB2;
SHOW DATABASES;
```



The `SHOW DATABASE` command in line 3 is used to display the list of databases present.

DROP DATABASE

The SQL **DROP DATABASE** statement is used to drop an existing database in SQL schema.

Syntax

The basic syntax of the `DROP DATABASE` statement is as follows:

```
DROP DATABASE DatabaseName;
```

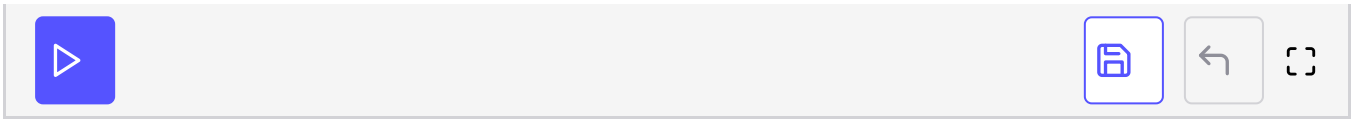
Example

If you want to delete an existing database, for example testDB1, then the `DROP DATABASE` statement would be as shown below:

```
DROP DATABASE testDB1;
```

Let's test this command in the code below:

```
CREATE DATABASE testDB1;
SHOW DATABASES;

DROP DATABASE testDB1;
SHOW DATABASES;
```



Line 4 in the above code is used to delete/drop the testDB1 database.

> Be careful when using this operation because deleting an existing database would result in a complete loss of information stored in the database.

USE database

When you have multiple databases in your SQL schema before starting your operation, you need to select the database where all the operations will be performed.

The SQL **USE DATABASE** statement is used to select any existing database in the SQL schema.

Syntax

The basic syntax of the `USE` statement is as shown below:

```
USE DatabaseName;
```

Example

Now, if you want to work with a database, for example testDB1, then you can execute the following SQL command and start working with it:

```
USE testDB1;
```

```
CREATE DATABASE testDB1;
CREATE DATABASE testDB2;
SHOW DATABASES;
```

```
USE testDB1;
```

If you want to work with a database in a separate file, the `USE` statement can be used to select the required database in the second file.

main.sql

temp.sql

```
CREATE DATABASE testDB1;
CREATE DATABASE testDB2;
```

Small quiz!

Q

Does the following query creates a new database called COMPANY and then uses it?

```
CREATE DATABASE COMPANY
USE COMPANY
```

COMPLETED 0%

1 of 1

In the next lesson, we will learn to create and delete tables (relations) in a database.

CREATE, DROP, and INSERT Table

In this lesson, we will take a look at three commands regarding relations/tables.

WE'LL COVER THE FOLLOWING

CREATE TABLE

Creating a basic table involves naming the table and defining its columns and the data type for each column.

The SQL **CREATE TABLE** statement is used to create a new table.

Syntax

The basic syntax of the `CREATE TABLE` statement is as follows:

```
CREATE TABLE table_name(
   column1 datatype,
   column2 datatype,
   column3 datatype,
```

```
   column3 datatype,

   .....

   columnN datatype,

   PRIMARY KEY(one or more columns)

);
```

`CREATE TABLE` is the keyword telling the database system what you want to do. The unique name or identifier for the table follows the `CREATE TABLE` statement.

Then, in brackets, comes the list defining each column in the table and what data type it is.

Example

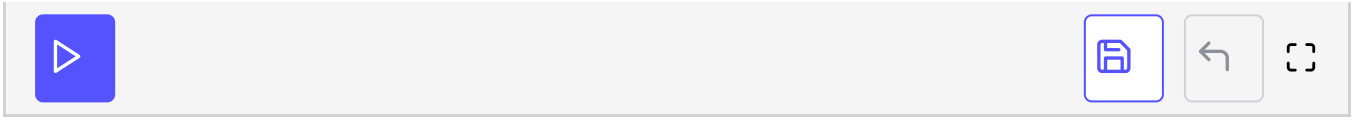
The following code block is an example which creates a CUSTOMERS table with ID as a primary key and **NOT NULL** is the constraint showing that these fields cannot be `NULL` while creating records in this table:

```
CREATE TABLE CUSTOMERS(
   ID        INT                    NOT NULL,
   NAME    VARCHAR (20)          NOT NULL,
   AGE       INT                    NOT NULL,
   ADDRESS     CHAR (25),
   SALARY       DECIMAL (18, 2),
   PRIMARY KEY    (ID)
);
```

You can verify that your table has been created successfully by using the **DESC command**:

```
CREATE TABLE CUSTOMERS(
```

```
  ID   INT              NOT NULL,
  NAME VARCHAR (20)     NOT NULL,
  AGE  INT              NOT NULL,

  ADDRESS  CHAR (25) ,
  SALARY   DECIMAL (18, 2),  /* The (18,2) simply means that we can have 18 digits with 2 of
  PRIMARY KEY (ID)
);

DESC CUSTOMERS;
```



The `DESC` command in **line 10** creates a table that contains information regarding the columns: name, data type, constraints.

Now, you have a CUSTOMERS table available in your database which you can use to store the required information related to customers.

DROP TABLE

The SQL **DROP TABLE** statement is used to remove a table definition and all the data, indexes, triggers, constraints and permission specifications for that table.

> You should be very careful while using this command because once a table is deleted, all the information available in that table will also be lost forever.

Syntax

The basic syntax of the `DROP TABLE` statement is as follows:

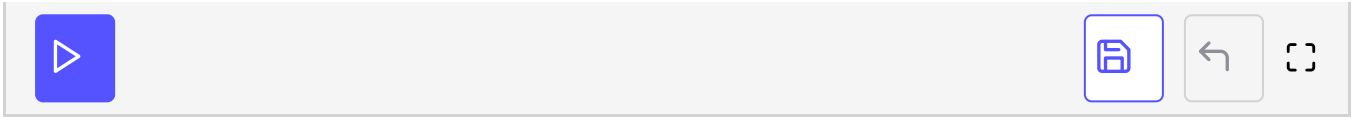
```
DROP TABLE table_name;
```

Example

Now let us delete the CUSTOMERS table we created above:

```
CREATE TABLE CUSTOMERS(
  ID   INT              NOT NULL,
  NAME VARCHAR (20)     NOT NULL,
  AGE  INT              NOT NULL,
  ADDRESS  CHAR (25) ,
  SALARY   DECIMAL (18, 2),  /* The (18,2) simply means that we can have 18 digits with 2 of
```

```
  PRIMARY KEY (ID)
);

DESC CUSTOMERS;

DROP TABLE CUSTOMERS;

DESC CUSTOMERS;
```



Now, if we try using the `DESC` command, we will get the above error which simply states that there is no table called CUSTOMERS in our ri_db database. Thus we have successfully dropped/deleted the CUSTOMERS table.

INSERT INTO

The SQL **INSERT INTO** statement is used to add new rows of data to a table in the database.

Syntax

There are two basic syntaxes of the `INSERT INTO` statement which are shown below.

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)

VALUES (value1, value2, value3,...valueN);
```

Here, column1, column2, column3,...columnN are the names of the columns in the table into which you want to insert the data.

You may not need to specify the column name(s) in the SQL query if you are adding values for all the columns in the table. But make sure the order of the values is the same as the columns in the table.

The SQL `INSERT INTO` syntax will be as follows:

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

Example

The following statements would create six records in the CUSTOMERS table:

First_Syntax | Second_Syntax

```
CREATE TABLE CUSTOMERS(
  ID   INT              NOT NULL,
  NAME VARCHAR (20)     NOT NULL,
  AGE  INT              NOT NULL,
  ADDRESS  CHAR (25) ,
  SALARY   DECIMAL (18, 2),  /* The (18,2) simply means that we can have 18 digits with 2 of
  PRIMARY KEY (ID)
);

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (1, 'Mark', 32, 'Texas', 50000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (2, 'John', 25, 'NY', 65000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (3, 'Emily', 23, 'Ohio', 20000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (4, 'Bill', 25, 'Chicago', 75000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (5, 'Tom', 27, 'Washington', 35000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (6, 'Jane', 22, 'Texas', 45000.00 );
```

The code written from **lines 10-26** will create the following table:

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50000.00
2	John	25	NY	65000.00
3	Emily	23	Ohio	20000.00
4	Bill	25	Chicago	75000.00
5	Tom	27	Washington	35000.00
6	Jane	22	Texas	45000.00

6	Jane	22	Texas	45000.00

Quick quiz!

Q

Is the following INSERT INTO statement correct?

```
INSERT INTO CUSTOMERS
VALUES (7, 'Troy', 'LA', 40000.00 );
```

COMPLETED 0%

1 of 1

In the next lesson, we will see how the `SELECT` statement is used to retrieve the columns of a table.

The SELECT Clause

In this lesson, we will take a look at the SELECT statement.

WE'LL COVER THE FOLLOWING

The SELECT clause

The SQL **SELECT** statement is used to fetch the data from a database table that returns this data in the form of a result table. These result tables are called **result-sets**.

Syntax

The basic syntax of the `SELECT` statement is as follows:

```
SELECT column1, column2, ... columnN FROM table_name;
```

Here, `SELECT` specifies the column1, column2... to be selected and the `FROM` clause specifies from which table these columns are to be selected.

If you want to fetch all the fields available in the table, then you can use the following syntax:

```
SELECT * FROM table_name;
```

Example

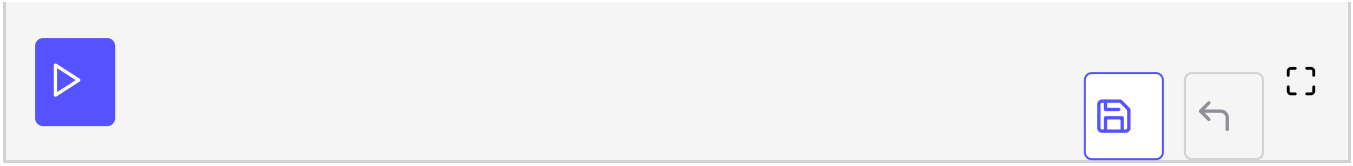
Consider the CUSTOMERS table we used in the last lesson:

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50000.00
2	John	25	NY	65000.00
3	Emily	23	Ohio	20000.00
4	Bill	25	Chicago	75000.00
5	Tom	27	Washington	35000.00
6	Jane	22	Texas	45000.00

Let's say we want to fetch the `ID`, `Name` and `Salary` fields of the customers available in the CUSTOMERS table. To do this we must specify these three column names after the `SELECT` statement. The following code shows how this is possible:

```
CREATE TABLE CUSTOMERS(
  ID   INT              NOT NULL,
  NAME VARCHAR (20)     NOT NULL,
  AGE  INT              NOT NULL,
  ADDRESS  CHAR (25) ,
  SALARY   DECIMAL (18, 2),  /* The (18,2) simply means that we can have 18 digits with 2 of
  PRIMARY KEY (ID)
);

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (1, 'Mark', 32, 'Texas', 50000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (2, 'John', 25, 'NY', 65000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (3, 'Emily', 23, 'Ohio', 20000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (4, 'Bill', 25, 'Chicago', 75000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (5, 'Tom', 27, 'Washington', 35000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (6, 'Jane', 22, 'Texas', 45000.00 );

SELECT ID, NAME, SALARY FROM CUSTOMERS;
```

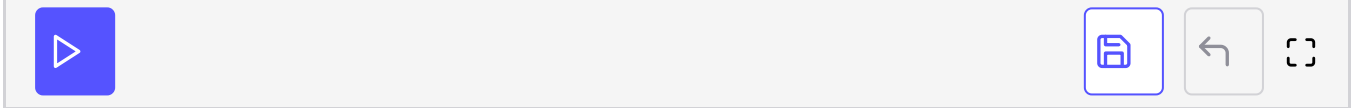
```
SELECT ID, NAME, SALARY FROM CUSTOMERS;
```



The `SELECT` statement on **line 28** is used to fetch the data in the specified columns.

Now let's fetch all of the columns using * after the `SELECT` clause in the CUSTOMERS table:

```
CREATE TABLE CUSTOMERS(
  ID   INT              NOT NULL,
  NAME VARCHAR (20)     NOT NULL,
  AGE  INT              NOT NULL,
  ADDRESS  CHAR (25) ,
  SALARY   DECIMAL (18, 2),  /* The (18,2) simply means that we can have 18 digits with 2 of
  PRIMARY KEY (ID)
);

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (1, 'Mark', 32, 'Texas', 50000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (2, 'John', 25, 'NY', 65000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (3, 'Emily', 23, 'Ohio', 20000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (4, 'Bill', 25, 'Chicago', 75000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (5, 'Tom', 27, 'Washington', 35000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (6, 'Jane', 22, 'Texas', 45000.00 );

SELECT * FROM CUSTOMERS;
```



Quick quiz!

Q

Which of the following SELECT statements will display the NAME, AGE and ADDRESS columns only?

In the next lesson, we will discuss how to use the `WHERE` clause.

TheWHERE Clause

In this lesson, we will learn how to use the WHERE clause in SQL to view specific data from table.

WE'LL COVER THE FOLLOWING

The WHERE clause

The SQL **WHERE** clause is used to specify a condition while fetching the data from a single table. If the given condition is satisfied, then those specific records are returned from the table.

Syntax

The basic syntax of the `SELECT` statement with the `WHERE` clause is as shown below:

```
SELECT column1, column2, ... columnN

FROM table_name

WHERE [condition];
```

You can specify a condition using the comparison or logical operators like >, <, =, `LIKE`, `NOT`, etc.

Example #1

Let's consider the CUSTOMERS table again.

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50000.00
2	John	25	NY	65000.00
3	Emily	23	Ohio	20000.00
4	Bill	25	Chicago	75000.00
5	Tom	27	Washington	35000.00
6	Jane	22	Texas	45000.00

Let's say we want to fetch the `ID`, `Name` and `Salary` fields from the CUSTOMERS table, provided that the salary of the customer is greater than $50,000.

So when we write our SQL query, will get the following result:

The WHERE clause will select those records where the salary is greater than 50000

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50000.00
2	John	25	NY	65000.00
3	Emily	23	Ohio	20000.00
4	Bill	25	Chicago	75000.00
5	Tom	27	Washington	35000.00
6	Jane	22	Texas	45000.00

The SELECT statement than retrieves the specified columns

ID	NAME	SALARY
2	John	65000.00
4	Bill	75000.00

2 of 2

The following code shows how to do this in SQL:

```
SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 50000;
```

Example #2

Let's consider another query, which would fetch all the fields from the CUSTOMERS table for a customer with the name John.

In this case, our query will produce the following result:

The WHERE clause will select those records where the customer name is John

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50000.00
2	John	25	NY	65000.00
3	Emily	23	Ohio	20000.00
4	Bill	25	Chicago	75000.00
5	Tom	27	Washington	35000.00
6	Jane	22	Texas	45000.00

The SELECT statement will return all the fields of the customer with the name John

ID	NAME	AGE	ADDRESS	SALARY
2	John	25	NY	65000.00

The SQL query is for this problem written below:

```
SELECT *
FROM CUSTOMERS
WHERE NAME = 'John';
```

Here, it is important to note that all the strings and characters should be inside single quotes (’’), whereas, numeric values should be given without any quotes.

Quick quiz!

Q

What will be the output of the following query?

```
SELECT NAME,  ADDRESS
FROM CUSTOMERS
WHERE ADDRESS = 'Texas';
```

In the next lesson, we will take a look a the `AND` & `OR` clause.

The AND & OR Clauses

In this lesson, we will discuss how can we combine multiple conditions in WHERE using the AND & OR operators.

WE'LL COVER THE FOLLOWING

The AND & OR clauses

The SQL **AND & OR** operators are used to combine multiple conditions in order to narrow data in an SQL statement. These two operators are called the **conjunctive operators**.

These operators provide a means to make multiple comparisons with different operators in the same SQL statement.

The AND Operator

The `AND` operator allows the existence of multiple conditions in a SQL statement's `WHERE` clause.

Syntax

The basic syntax of the `AND` operator with a `WHERE` clause is as follows:

```
SELECT column1, column2, ... columnN
```

```
FROM table_name


WHERE [condition1] AND [condition2]...AND [conditionN];
```

You can combine N number of conditions using the `AND` operator. For an action to be taken by the SQL statement, whether it be a transaction or a query, all conditions separated by the `AND` must be `TRUE`.

Example

In this example, we will retrieve the `ID`, `Name` and `Salary` fields from the CUSTOMERS table, where the salary is greater than $20,000 (inclusive) and the age is less than 25 years.

The steps needed to solve this problem are highlighted below:

The WHERE clause will return those customers that have both salary greater than 20000 and are less than 20 years old

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50000.00
2	John	25	NY	65000.00
3	Emily	23	Ohio	20000.00
4	Bill	25	Chicago	75000.00
5	Tom	27	Washington	35000.00
6	Jane	22	Texas	45000.00

The SELECT clause will return the data from the specified columns

ID	NAME	SALARY
3	Emily	20000.00
6	Jane	45000.00

2 of 2

The following code shows how to do this in SQL:

```
SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY >= 20000 AND age < 25;
```

The OR Operator

The `OR` operator is used to combine multiple conditions in a SQL statement's `WHERE` clause.

Syntax

The basic syntax of the `OR` operator with a `WHERE` clause is as follows:

```
SELECT column1, column2, ... columnN

FROM table_name
```

```
FROM table_name

WHERE [condition1] OR [condition2]...OR [conditionN];
```

You can combine N number of conditions using the `OR` operator. For an action to be taken by the SQL statement, whether it be a transaction or query, only ONE of the conditions separated by the `OR` can be `TRUE`.

Example

Consider the following query, which will fetch the `ID`, `Name` and `Salary` fields from the CUSTOMERS table, where the salary is greater than $50,000 or the age is less than 25 years.

The following slides show the steps needed to solve this problem:

The WHERE clause will return those customers that either have salary greater than 20000 or they are less than 20 years old

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50000.00
2	John	25	NY	65000.00
3	Emily	23	Ohio	20000.00
4	Bill	25	Chicago	75000.00
5	Tom	27	Washington	35000.00
6	Jane	22	Texas	45000.00

The SELECT clause will return the data from the specified columns

ID	NAME	SALARY
2	John	65000.00
3	Emily	20000.00
4	Bill	75000.00
6	Jane	45000.00

The following code shows how to do this in SQL:

```
SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 50000 OR age < 25;
```

Quick quiz!

Q

Which of the following query will return the customers who have SALARY between 40000 and 60000?

In the next lesson, we will learn about SQL aggregate functions.

Aggregate Functions in SQL

In this lesson, we will learn about the different aggregate functions available in SQL.

WE'LL COVER THE FOLLOWING

Aggregate functions in SQL

In database management, an aggregate function is a function where the values of multiple rows are grouped together to form a single value of more significant meaning.

We will discuss the following in this lesson:

- `COUNT()`

- `SUM()`
- `AVG()`
- `MIN()`
- `MAX()`

Again we will be using the CUSTOMERS table.

The COUNT function

The `COUNT()` function returns the number of rows that match a specified criterion.

Syntax

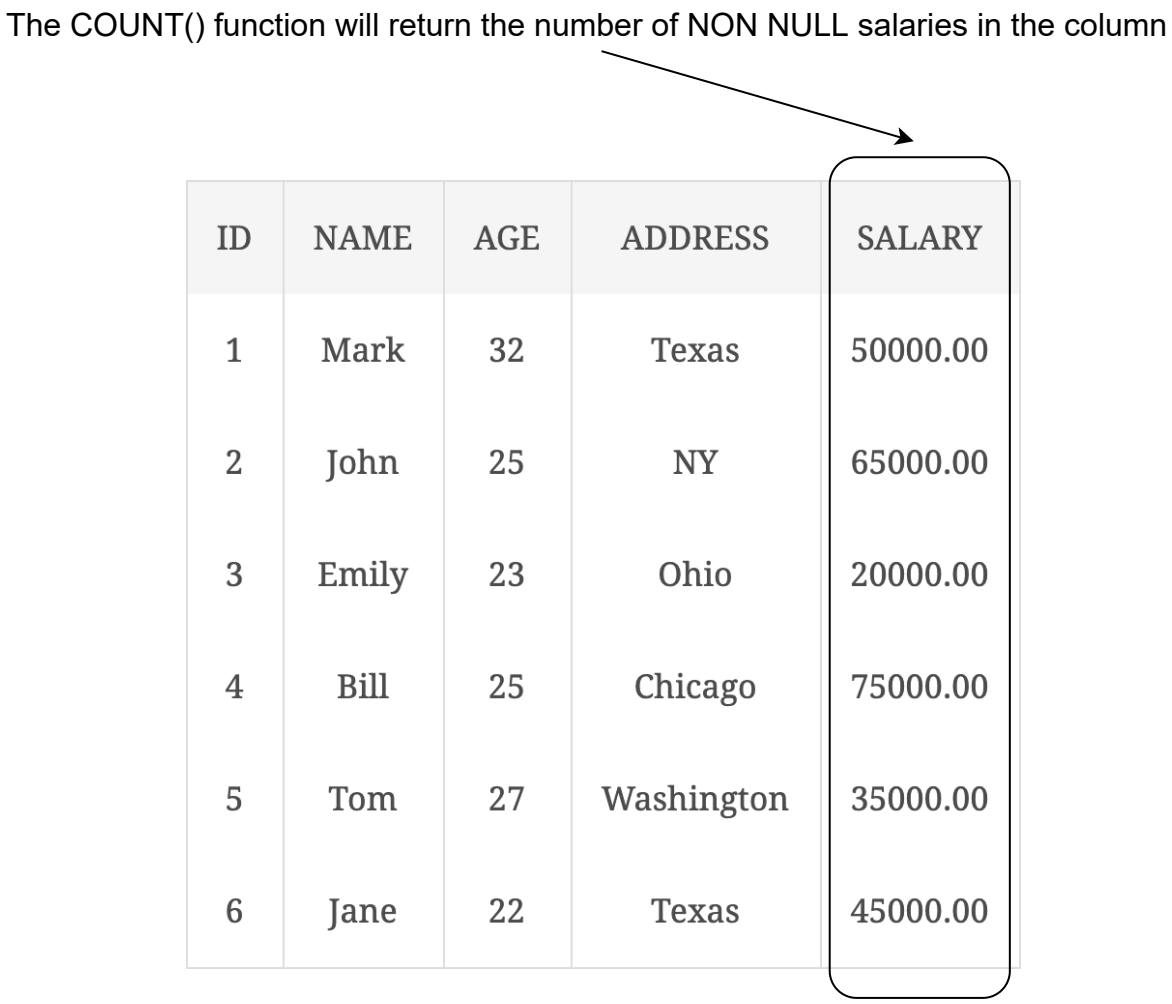
The syntax for the `COUNT()` function is as follows:

```
SELECT COUNT(column_name)

FROM table_name

WHERE condition;
```

This query will return the number of `Non-Null` values in the specified column.

Example

Let's say we apply the COUNT function to the `salary` column:



ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50000.00
2	John	25	NY	65000.00
3	Emily	23	Ohio	20000.00
4	Bill	25	Chicago	75000.00
5	Tom	27	Washington	35000.00
6	Jane	22	Texas	45000.00

The following code shows the SQL query:

```
SELECT COUNT(SALARY)
FROM CUSTOMERS;
```

As we can see it returned the number of `Non-Null` values over the column salary i.e, 6.

The SUM function

The `SUM()` function returns the total sum of a numeric column.

Syntax

The syntax for the SUM() function is as follows:

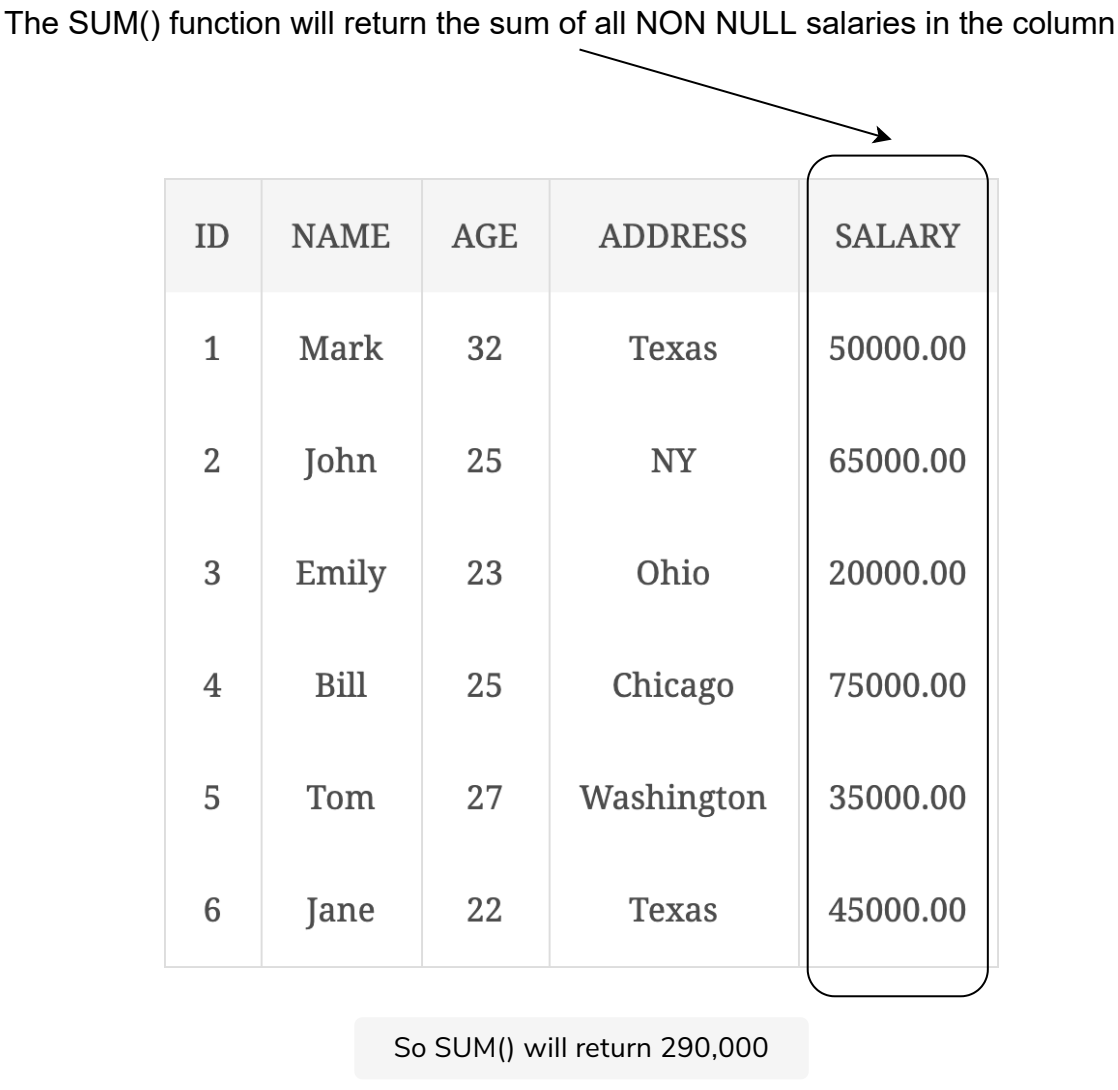
```
SELECT SUM(column_name)
```

```
FROM table_name



WHERE condition;
```

This query will return the sum of all `Non-Null` values in a particular column.

Example

Let's say we apply the `SUM` function to the `salary` column:



ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50000.00
2	John	25	NY	65000.00
3	Emily	23	Ohio	20000.00
4	Bill	25	Chicago	75000.00
5	Tom	27	Washington	35000.00
6	Jane	22	Texas	45000.00

The following code shows the SQL query:

```
SELECT SUM(SALARY)
FROM CUSTOMERS;
```

As we can see in the output above, the sum of all `Non-Null` values in the salary column is 290,000.

The AVG function

The `AVG()` function returns the average value of a numeric column.

Syntax

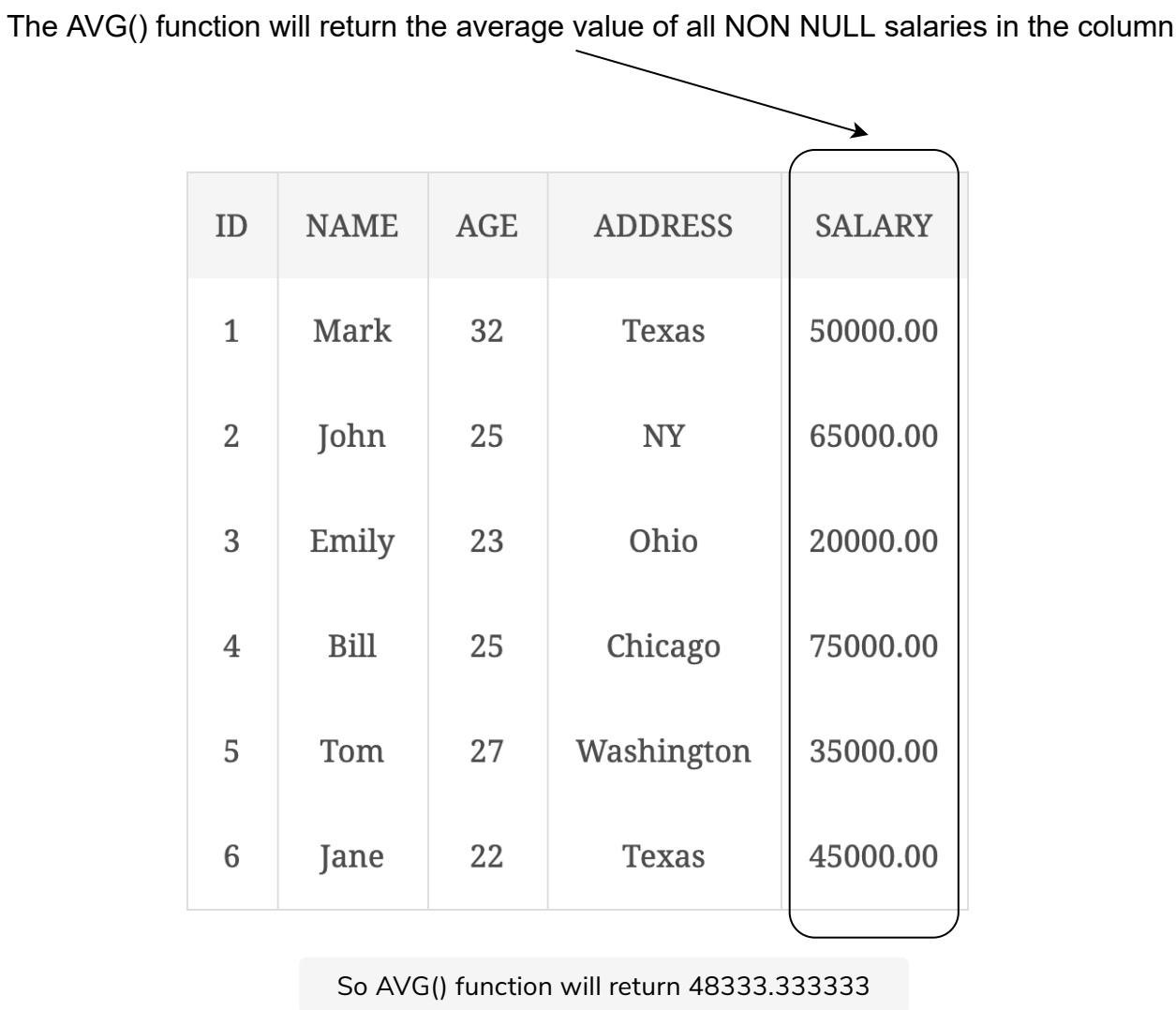
The syntax for the `AVG()` function is as follows:

```
SELECT AVG(column_name)

FROM table_name

WHERE condition;
```

This query will return the average of all `Non-Null` values in a particular column.

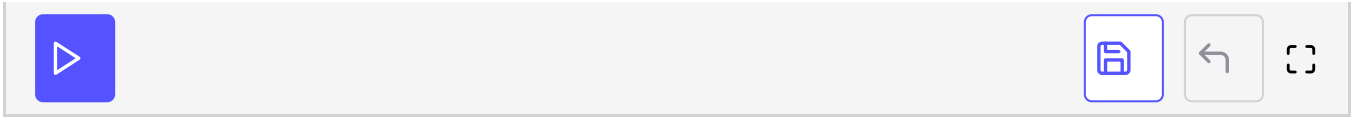
Example

Let's say we apply the `AVG` function to the `salary` column:



The following code shows the SQL query:

```
SELECT AVG(SALARY)
FROM CUSTOMERS;
```



As we can see, it returned the average of `Non-Null` values of the column salary, i.e. 48333.33.

The MAX function

The `MAX()` function returns the largest value of the selected column.

Syntax

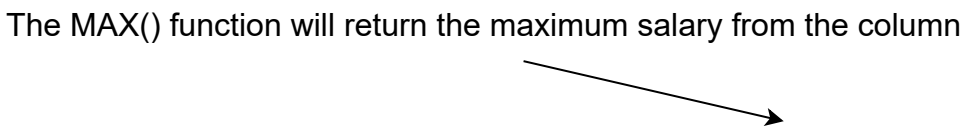
The syntax for the `MAX()` function is as follows:

```
SELECT MAX(column_name)

FROM table_name

WHERE condition;
```

This query will return the max of all `Non-Null` values in a particular column.

Example

Let's say we want to find the highest salary in the CUSTOMERS table:



ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50000.00
2	John	25	NY	65000.00
3	Emily	23	Ohio	20000.00
4	Bill	25	Chicago	75000.00
5	Tom	27	Washington	35000.00
6	Jane	22	Texas	45000.00

So MAX() function will return 75000.00

The following code shows the SQL query:

```
SELECT MAX(SALARY)
FROM CUSTOMERS
```

The MIN function

The `MIN()` function returns the smallest value in the selected column.

Syntax

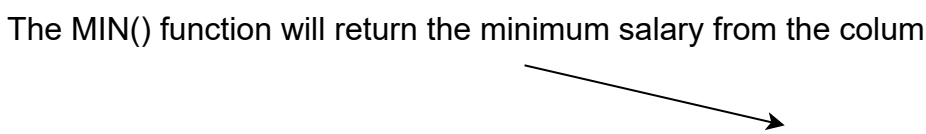
The syntax for the `MIN()` function is as follows:

```
SELECT MIN(column_name)

FROM table_name

WHERE condition;
```

This query will return the min of all `Non-Null` values in a particular column.

Example

Let's say we want to find the lowest salary in the CUSTOMERS table:



ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50000.00
2	John	25	NY	65000.00
3	Emily	23	Ohio	20000.00
4	Bill	25	Chicago	75000.00
5	Tom	27	Washington	35000.00
6	Jane	22	Texas	45000.00

So MAX() function will return 75000.00

The following code shows the SQL query:

```
SELECT MIN(SALARY)
FROM CUSTOMERS;
```

Quick quiz!

Q

Which of the following SQL queries will return the youngest person in the CUSTOMERS table?

In the next lesson, we will discuss two important clauses: ORDER BY and GROUP BY.

ORDER BY & GROUP BY

In this lesson, we will learn about the ORDER BY and GROUP BY clauses.

WE'LL COVER THE FOLLOWING

The ORDER BY clause

The SQL **ORDER BY** clause is used to sort the data of one or more columns in ascending or descending order. Some databases sort the query results in ascending order by default.

Syntax

The basic syntax of the `ORDER BY` clause is as follows:

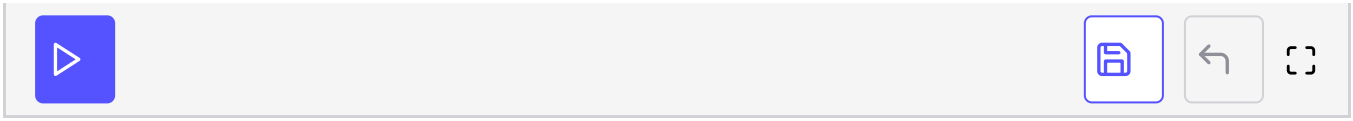
```
SELECT column-list

FROM table_name

WHERE condition

ORDER BY column1, column2, .. columnN;
```

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort is in the column-list.

Example

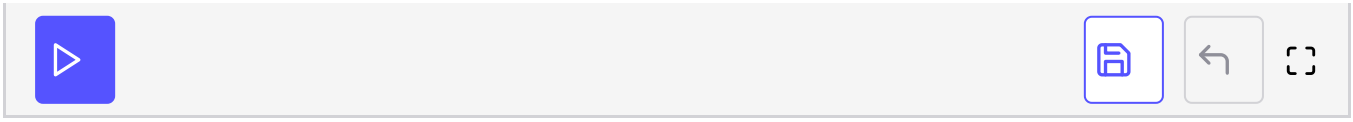
We will sort the CUSTOMERS table in ascending order by the `NAME` column:

```
SELECT *
FROM CUSTOMERS
ORDER BY NAME;
```



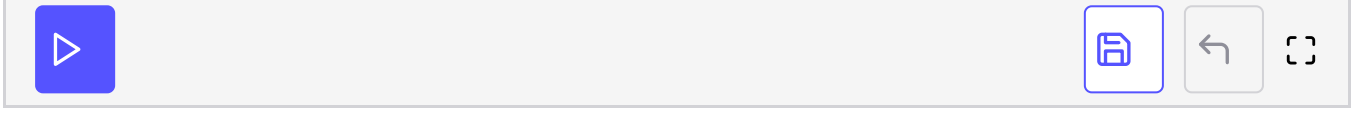
Now let's say we want to sort the list according to `NAME` but in descending order. The code below depicts this:

```
SELECT *
FROM CUSTOMERS
ORDER BY NAME DESC;
```



If we don't write anything after the `ORDER BY` clause, then the column will be sorted in ascending order by default. However, we can also specify that we want to sort the list in ascending order by using the `ASC` keyword. The code below depicts this:

```
SELECT *
FROM CUSTOMERS
ORDER BY NAME ASC;
```



The GROUP BY clause

The SQL **GROUP BY** clause is used in collaboration with the `SELECT` statement to arrange identical data into groups. The `GROUP BY` clause follows the `WHERE` clause in a `SELECT` statement and precedes the `ORDER BY` clause.

Syntax

The basic syntax of a `GROUP BY` clause is shown below. The `GROUP BY` clause must follow the conditions in the `WHERE` clause and must precede the `ORDER BY`

clause if one is used.

```
SELECT column1, column2 ... columnN

FROM table_name

WHERE conditions

GROUP BY column1, column2 ... columnN

ORDER BY column1, column2 ... columnN;
```

Example

Consider the CUSTOMERS table below but with a few changes:

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	Mark	23	LA	77,000
3	John	25	NY	65,000
4	Emily	23	Ohio	20,000
5	John	31	Arizona	54,000
6	Bill	25	Chicago	75,000
7	Bill	28	Florida	31,000
8	Emily	29	Michigan	43,000
9	Tom	27	Washington	35,000
10	Jane	22	Texas	45,0000

As you can see, there are duplicate names in the table above.

If you want to know the total amount earned by customers with the same name, then the `GROUP BY` query will return the following result:

As we want to sum the salaries of people with same names. We will apply the GROUP BY query on the NAME column so we can group together the people with same name.

So these two people with the name Mark will be grouped together and we will simply add their salaries

The same will be done for the different groups of people with same names.

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	Mark	23	LA	77,000
3	John	25	NY	65,000
4	Emily	23	Ohio	20,000
5	John	31	Arizona	54,000
6	Bill	25	Chicago	75,000
7	Bill	28	Florida	31,000
8	Emily	29	Michigan	43,000
9	Tom	27	Washington	35,000
10	Jane	22	Texas	45,0000

As we mentioned in the previous slide, each group of people with the same name will have their salaries totalled. These totals are presented in a seperate column.

NAME	SUM(SALARY)
Bill	106000.00
Emily	63000.00
Jane	45000.00
John	119000.00
Mark	127000.00
Tom	35000.00

The code for the `GROUP BY` query is written below:

```
SELECT NAME, SUM(SALARY)
FROM CUSTOMERS
GROUP BY NAME
ORDER BY NAME;
```

In **line 3**, the `GROUP BY` statement groups the customers based on their names and then the `SUM()` function is applied over the SALARY column so we get the total salary per customer group.

Quick quiz!

Q

What will be the ouput of the following query?

```
SELECT NAME, MIN(AGE)
FROM CUSTOMERS
GROUP BY NAME;
```

COMPLETED 0%

1 of 1

In the next lesson, we will learn about the HAVING clause.

The HAVING Clause

In this lesson, we will learn about the HAVING clause.

WE'LL COVER THE FOLLOWING

The HAVING clause

The **HAVING** clause is utilized in SQL as a conditional clause with the `GROUP BY` clause. This conditional clause only returns rows where aggregate function results are matched with given conditions.

The `HAVING` clause was added to SQL because the `WHERE` keyword could not be used with aggregate functions.

Syntax

The basic syntax of the `HAVING` clause is as follows:

```
SELECT column1, column2, ... columnN

FROM table_name

WHERE [ conditions ]

GROUP BY column1, column2, ... columnN

HAVING [ conditions ]

ORDER BY column1, column2, ... columnN;
```

As you can see, the `HAVING` clause must follow the `GROUP BY` clause in a query and must also precede the `ORDER BY` clause if used.

Example

Consider the CUSTOMERS table below but with a few changes:

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	Jeff	23	LA	77,000
3	John	25	NY	65,000
4	Emily	23	Ohio	20,000
5	John	31	Texas	54,000
6	Bill	25	Texas	75,000
7	Bob	28	NY	31,000
8	Elyse	29	Ohio	43,000
9	Tom	27	Washington	35,000
10	Jane	22	NY	45,0000

As you can see, there are many customers that live at the same `ADDRESS` (i.e. live in the same state).

We want to write a SQL statement that returns the number of customers in each state, but only if that state has more than 2 customers:

As we want to count the number of people living in the same state. We will apply the GROUP BY query on the ADDRESS column so we can group together the people who live in the same state.

We see these three people live in Texas, so they will be grouped together

The same will be done for the different groups of people living in the same states.

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	Jeff	23	LA	77,000
3	John	25	NY	65,000
4	Emily	23	Ohio	20,000
5	John	31	Texas	54,000
6	Bill	25	Texas	75,000
7	Bob	28	NY	31,000
8	Elyse	29	Ohio	43,000
9	Tom	27	Washington	35,000
10	Jane	22	NY	45,0000

After being grouped together, the COUNT() function will count the number of people in each group. Then the HAVING clause will only return those ADDRESS that have count greater then two

ADDRESS	COUNT(ID)
NY	3
Texas	3

The code for the above query is written below:

```
SELECT ADDRESS, COUNT(ID)
FROM CUSTOMERS

GROUP BY ADDRESS
HAVING COUNT(ID) > 2;
```

In **line 3**, the GROUP BY statement groups the customers based on their `ADDRESS` and then the `HAVING` clause in **line 4** checks to see if the number of customers living in this state is greater than two using the `COUNT()` function.

Quick quiz!

Q

Will the following SQL statement will return those ADDRESS (i.e. states) that have customers who earn a combine total greater than 80000?

```
SELECT ADDRESS, SUM(SALARY)
FROM CUSTOMERS
GROUP BY ADDRESS
HAVING SUM(SALARY) > 80000;
```

COMPLETED 0%

1 of 1

In the next lesson, we will learn to assign aliases to columns and tables.

Alias Syntax

In this lesson, we will learn about assigning aliases to tables and columns.

WE'LL COVER THE FOLLOWING

Alias syntax

You can rename a table or a column temporarily by giving another name known as an **Alias**. The use of table aliases is to rename a table in a specific SQL statement. Column aliases are used to rename a table's columns for a particular SQL query. This renaming is a temporary change and the actual table/column name does not change in the database.

Syntax of a table alias

The basic syntax of a `table alias` is as follows:

```
SELECT column1, column2 ... columnN

FROM table_name AS alias_name

WHERE condition;
```

Example

For our example, we will be using the following tables:

Customer Table

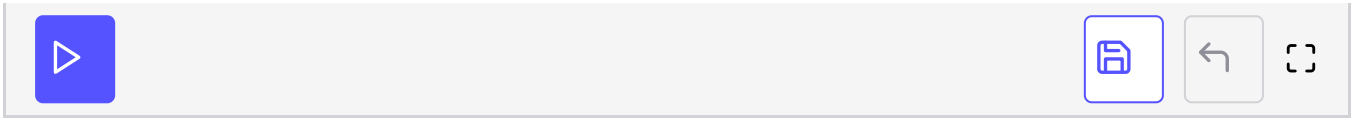
ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	John	25	NY	65,000
3	Emily	23	Ohio	20,000
4	Bill	25	Chicago	75,000
5	Tom	27	Washington	35,000
6	Jane	22	Texas	45,000

Orders Table

ORDER_ID	DATE	CUSTOMER_ID	AMOUNT
100	2019-09-08	2	5000
101	2019-08-20	5	3000
102	2019-05-12	1	1000
103	2019-02-02	2	2000

```
/* This is the same table we created in the previous lessons.*/
CREATE TABLE CUSTOMERS(
  ID   INT              NOT NULL,
  NAME VARCHAR (20)     NOT NULL,
  AGE  INT              NOT NULL,
  ADDRESS  CHAR (25) ,
  SALARY   DECIMAL (18, 2),  /* The (18,2) simply means that we can have 18 digits with 2 of
  PRIMARY KEY (ID)
);

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (1, 'Mark', 32, 'Texas', 50000.00 );
```

```
INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (2, 'John', 25, 'NY', 65000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (3, 'Emily', 23, 'Ohio', 20000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (4, 'Bill', 25, 'Chicago', 75000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (5, 'Tom', 27, 'Washington', 35000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (6, 'Jane', 22, 'Texas', 45000.00 );

/*We will now create the ORDERS table*/
CREATE TABLE ORDERS(
  ORDER_ID   INT        NOT NULL,
  DATE VARCHAR (20)     NOT NULL,
  CUSTOMER_ID  INT      NOT NULL,
  AMOUNT   INT,
  PRIMARY KEY (ORDER_ID),
  FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMERS(ID) /* We must specify the table to which th
);

INSERT INTO ORDERS (ORDER_ID, DATE, CUSTOMER_ID, AMOUNT)
VALUES (100, '2019-09-08', 2, 5000 );

INSERT INTO ORDERS (ORDER_ID, DATE, CUSTOMER_ID, AMOUNT)
VALUES (101, '2019-08-20', 5, 3000 );

INSERT INTO ORDERS (ORDER_ID, DATE, CUSTOMER_ID, AMOUNT)
VALUES (102, '2019-05-12', 1, 1000 );

INSERT INTO ORDERS (ORDER_ID, DATE, CUSTOMER_ID, AMOUNT)
VALUES (103, '2019-02-02', 2, 2000 );

SELECT C.ID, C.NAME, C.AGE, O.AMOUNT
FROM CUSTOMERS AS C, ORDERS AS O
WHERE  C.ID = O.CUSTOMER_ID;
```



As you can see in the code above, we temporarily assigned a new alias `C` to the CUSTOMERS table and `O` to the ORDERS table.

Furthermore, when working with two tables, we need to specify the table name from which the column is derived, therefore having short aliases helps us to avoid writing large names before each column name. Also, sometimes tables can have the same column names so specifying the table name before the column name helps to avoid confusion regarding which table we are referring to.

In the highlighted lines above, the query simply displays the relevant fields we have selected based on the condition that `ID` in the CUSTOMERS table is equal to `ORDER_ID` in ORDERS table. So the result-set is the group of people who have placed orders.

Syntax for a column alias

The basic syntax of a `column alias` is as follows:

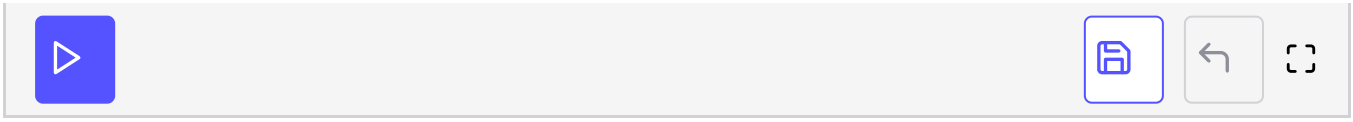
```
SELECT column_name AS alias_name

FROM table_name

WHERE condition;
```

EXAMPLE

Following is the usage of a `column alias`.

```
/* This is the same table we created in the previous lessons.*/
CREATE TABLE CUSTOMERS(
  ID   INT              NOT NULL,
  NAME VARCHAR (20)     NOT NULL,
  AGE  INT              NOT NULL,
  ADDRESS  CHAR (25) ,
  SALARY   DECIMAL (18, 2),  /* The (18,2) simply means that we can have 18 digits with 2 of
  PRIMARY KEY (ID)
);

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (1, 'Mark', 32, 'Texas', 50000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (2, 'John', 25, 'NY', 65000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (3, 'Emily', 23, 'Ohio', 20000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (4, 'Bill', 25, 'Chicago', 75000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (5, 'Tom', 27, 'Washington', 35000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (6, 'Jane', 22, 'Texas', 45000.00 );

/*We will now create the ORDERS table*/
CREATE TABLE ORDERS(
  ORDER_ID   INT        NOT NULL,
  DATE VARCHAR (20)     NOT NULL,
  CUSTOMER_ID  INT      NOT NULL,
  AMOUNT   INT,
```

```
  PRIMARY KEY (ORDER_ID),
  FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMERS(ID) /* We must specify the table to which th
);

INSERT INTO ORDERS (ORDER_ID, DATE, CUSTOMER_ID, AMOUNT)
VALUES (100, '2019-09-08', 2, 5000 );

INSERT INTO ORDERS (ORDER_ID, DATE, CUSTOMER_ID, AMOUNT)
VALUES (101, '2019-08-20', 5, 3000 );

INSERT INTO ORDERS (ORDER_ID, DATE, CUSTOMER_ID, AMOUNT)
VALUES (102, '2019-05-12', 1, 1000 );

INSERT INTO ORDERS (ORDER_ID, DATE, CUSTOMER_ID, AMOUNT)
VALUES (103, '2019-02-02', 2, 2000 );

SELECT  ID AS CUSTOMER_ID, NAME AS CUSTOMER_NAME
FROM CUSTOMERS
WHERE SALARY > 40000.00;
```



As you can see in the output above, the column names have changed.

A `column alias` is particularly useful when we want to change the name of a column to one that is easier to understand for the user.

Quick quiz!

Q

The following query will change the name of the column `ID` to `CUSTOMER_ID`

```
SELECT  ID , NAME AS CUSTOMER_NAME
FROM CUSTOMERS AS CUST
```

COMPLETED 0%

1 of 1

In the next lesson, we will learn to combine two different tables using joins.

SQL Joins

In this lesson, we will highlight the different types of joins in SQL.

WE'LL COVER THE FOLLOWING

SQL JOIN

A `JOIN` clause is used to combine rows from two or more tables, based on a common column.

We will be using the CUSTOMER and ORDER tables as shown below:

Customer Table

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	John	25	NY	65,000
3	Emily	23	Ohio	20,000
4	Bill	25	Chicago	75,000
5	Tom	27	Washington	35,000
6	Jane	22	Texas	45,000

Orders Table

ORDER_ID	DATE	CUSTOMER_ID	AMOUNT
100	2019-09-08	2	5000
101	2019-08-20	5	3000
102	2019-05-12	1	1000
103	2019-02-02	2	2000

Notice that the `CUSTOMER_ID` in the ORDER table references `ID` in the CUSTOMER table.

Now, what if we need to query something that is the combination of information in both tables?

For example, we want to:

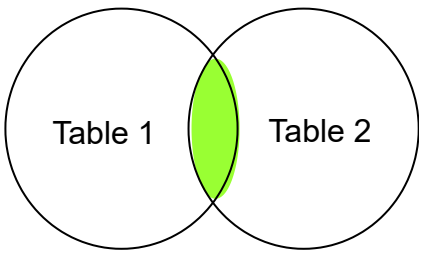
- Find information on customers who ordered an item.
- Find the number of customers who ordered a certain item.
- Find the address of a customer in order to dispatch the order.

The joins in SQL can help you do that using the `JOIN` clause.

Different types of SQL JOINs

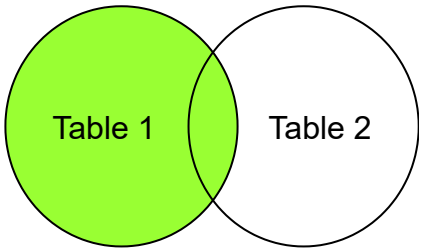
Here are the three different types of the JOINs we will be discussing in this chapter:

- **INNER JOIN / JOIN**: Returns records that have matching values in both tables.



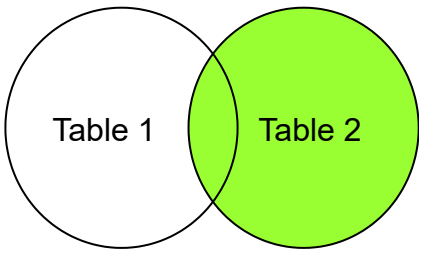
The values that are a match in both tables will be returned in the result-set.

- **LEFT JOIN/ LEFT OUTER JOIN**: Returns all records from the left table, and the matched records from the right table.



All the values from the left table and matching values will be returned in the result-set.

- **RIGHT JOIN/ RIGHT OUTER**: Returns all records from the right table, and the matched records from the left table.



All the values from the right table and matching values will be returned in the result-set.

In the next lesson, we will discuss the inner join in more detail.

INNER JOIN

In this lesson, we will study the INNER JOIN in SQL.

WE'LL COVER THE FOLLOWING

INNER JOIN

The **INNER JOIN** keyword selects records that have matching values in both tables.

Syntax

The basic syntax of the `INNER JOIN` is as follows:

```
SELECT table1.column1, table2.column2 ...

FROM table1

INNER JOIN table2

ON table1.common_field = table2.common_field;
```

Example

We will be using the CUSTOMERS and ORDERS tables as defined in the previous lesson.

Let's say we want to retrieve the information of only those customers that have placed an order. This can be done by joining the two tables:

The CUSTOMERS table contains information regarding the customers, while the ORDERS table contains information regarding orders placed by customers. So as we want information from both the tables we will join them.

Customer Table

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	John	25	NY	65,000
3	Emily	23	Ohio	20,000
4	Bill	25	Chicago	75,000
5	Tom	27	Washington	35,000
6	Jane	22	Texas	45,000

Orders Table

Order_Id	Date	Customer_Id	Amount
100	2019-09-08	2	5000
101	2019-08-20	5	3000
102	2019-05-12	1	1000
103	2019-02-02	2	2000

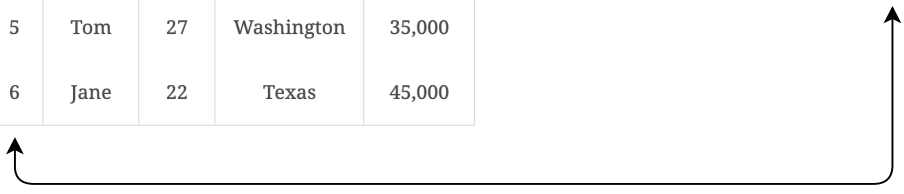
We will use INNER JOIN in this case. To use INNER JOIN we must specify a common column between the two tables. We can see that the Customer_Id column in ORDERS refers to the ID column in CUSTOMERS.

Customer Table

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	John	25	NY	65,000
3	Emily	23	Ohio	20,000
4	Bill	25	Chicago	75,000
5	Tom	27	Washington	35,000
6	Jane	22	Texas	45,000

Orders Table

Order_Id	Date	Customer_Id	Amount
100	2019-09-08	2	5000
101	2019-08-20	5	3000
102	2019-05-12	1	1000
103	2019-02-02	2	2000



The column that links the two tables

We use INNER JOIN because we want the information of only those customers who have placed an order, thus INNER JOIN will be used to retrieve only the matching records in both tables.

Customer Table

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	John	25	NY	65,000
3	Emily	23	Ohio	20,000
4	Bill	25	Chicago	75,000
5	Tom	27	Washington	35,000
6	Jane	22	Texas	45,000

Orders Table

Order_Id	Date	Customer_Id	Amount
100	2019-09-08	2	5000
101	2019-08-20	5	3000
102	2019-05-12	1	1000
103	2019-02-02	2	2000

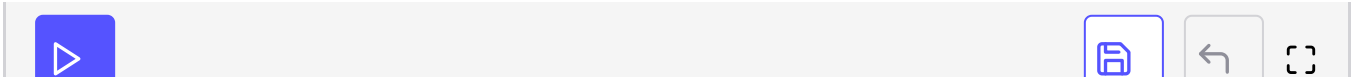
As we can see the INNER JOIN will return those records where the customer IDs match in both the tables.

After the INNER JOIN returns the matching records, we can use the SELECT statement to dislay the required columns only. Let's say we want the ID, NAME, AMOUNT and DATE columns only.The final resulant table is shown below:

ID	NAME	AMOUNT	DATE
2	John	5000	2019-09-08
5	Tom	3000	2019-08-20
1	Mark	1000	2019-05-12
2	John	2000	2019-02-02

The following code will show you how to join the two tables:

```
/* This is the same table we created in the previous lessons.*/
CREATE TABLE CUSTOMERS(
  ID   INT              NOT NULL,
  NAME VARCHAR (20)     NOT NULL,
  AGE  INT              NOT NULL,
  ADDRESS  CHAR (25) ,
  SALARY   DECIMAL (18, 2),  /* The (18,2) simply means that we can have 18 digits with 2 of
  PRIMARY KEY (ID)
);

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (1, 'Mark', 32, 'Texas', 50000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (2, 'John', 25, 'NY', 65000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (3, 'Emily', 23, 'Ohio', 20000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (4, 'Bill', 25, 'Chicago', 75000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (5, 'Tom', 27, 'Washington', 35000.00 );

INSERT INTO CUSTOMERS (ID, NAME, AGE, ADDRESS, SALARY)
VALUES (6, 'Jane', 22, 'Texas', 45000.00 );

/*This is the same ORDERS table we created in previous lectures.*/
CREATE TABLE ORDERS(
  ORDER_ID   INT        NOT NULL,
  DATE VARCHAR (20)     NOT NULL,
  CUSTOMER_ID  INT      NOT NULL,
  AMOUNT   INT,
  PRIMARY KEY (ORDER_ID),
  FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMERS(ID) /* We must specify the table to which th
);

INSERT INTO ORDERS (ORDER_ID, DATE, CUSTOMER_ID, AMOUNT)
VALUES (100, '2019-09-08', 2, 5000 );

INSERT INTO ORDERS (ORDER_ID, DATE, CUSTOMER_ID, AMOUNT)
VALUES (101, '2019-08-20', 5, 3000 );

INSERT INTO ORDERS (ORDER_ID, DATE, CUSTOMER_ID, AMOUNT)
VALUES (102, '2019-05-12', 1, 1000 );

INSERT INTO ORDERS (ORDER_ID, DATE, CUSTOMER_ID, AMOUNT)
VALUES (103, '2019-02-02', 2, 2000 );


SELECT  CUSTOMERS.ID, CUSTOMERS.NAME, ORDERS.AMOUNT, ORDERS.DATE
FROM CUSTOMERS
INNER JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```



> Note: The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are records in the “Orders” table that do not have matches in “Customers”, these orders will not be shown.

That is why we don’t see Emily, Bill or Jane in the result-set; they have not placed any orders.

Quick quiz!

Q

Which of the following queries will return the NAME and AGE of a customer along with the DATE they placed an order?

In the next lesson, we will take a look at the LEFT JOIN keyword.

LEFT JOIN

In this lesson, we will discuss the LEFT JOIN keyword.

WE'LL COVER THE FOLLOWING

LEFT JOIN

The **LEFT JOIN** keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is `NULL` from the right side if there is no match.

Syntax

```
SELECT table1.column1, table2.column2...

FROM table1

LEFT JOIN table2

ON table1.common_field = table2.common_field;
```

> Note: In some databases, LEFT JOIN is called LEFT OUTER JOIN.

Example

We want to select all customers and any orders they might have placed:

The CUSTOMERS table contains information regarding the customers, while the ORDERS table contains information regarding orders placed by customers. As we want the customer infromation even if they have not placed an order, so we will use LEFT JOIN.

Customer Table

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	John	25	NY	65,000
3	Emily	23	Ohio	20,000
4	Bill	25	Chicago	75,000
5	Tom	27	Washington	35,000
6	Jane	22	Texas	45,000

Orders Table

Order_Id	Date	Customer_Id	Amount
100	2019-09-08	2	5000
101	2019-08-20	5	3000
102	2019-05-12	1	1000
103	2019-02-02	2	2000

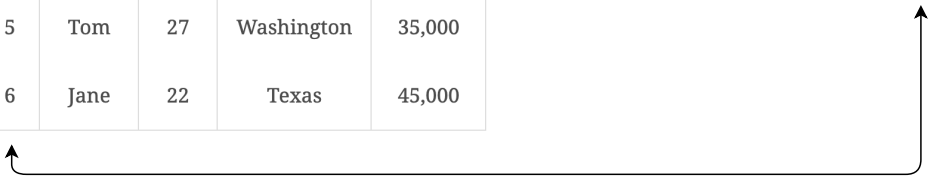
Similar to INNER JOIN, if we want to use LEFT JOIN we must specify a common column between the two tables.

Customer Table

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	John	25	NY	65,000
3	Emily	23	Ohio	20,000
4	Bill	25	Chicago	75,000
5	Tom	27	Washington	35,000
6	Jane	22	Texas	45,000

Orders Table

Order_Id	Date	Customer_Id	Amount
100	2019-09-08	2	5000
101	2019-08-20	5	3000
102	2019-05-12	1	1000
103	2019-02-02	2	2000



This column is common between the two tables

The LEFT JOIN returns all of the records in the CUSTOMERS table along with the matched reords in the ORDERS table. If there is no match i.e. customer has not placed an order that particular record will have NULL values.

ID	NAME	AMOUNT	DATE
1	Mark	1000	2019-05-12
2	John	5000	2019-09-08
2	John	2000	2019-02-02
3	Emily	NULL	NULL
4	Bill	NULL	NULL
5	Tom	3000	2019-08-20
6	Jane	NULL	NULL

Like before we use SELECT to display the desired columns only.

3 of 3

The SQL query to retrieve all customers whether or not they have placed an order:

```
SELECT  CUSTOMERS.ID, CUSTOMERS.NAME, ORDERS.AMOUNT, ORDERS.DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
ORDER BY CUSTOMERS.ID;
```

As you can see, the LEFT JOIN keyword returns all records from the left table (Customers), even if there are no matches in the right table (Orders).

Quick quiz!

Q

Will the following query return the customer NAME and ADDRESS along with the AMOUNT they purchased?

```
SELECT CUSTOMERS.NAME, CUSTOMERS.ADDRESS , ORDERS.AMOUNT
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```


In the next lesson, we will discuss the RIGHT JOIN keyword.

RIGHT JOIN

In this lesson, we will discuss the RIGHT JOIN keyword.

WE'LL COVER THE FOLLOWING

RIGHT JOIN

The **RIGHT JOIN** keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is `NULL` from the left side when there is no match.

Syntax

```
SELECT table1.column1, table2.column2...

 FROM table1

RIGHT JOIN table2

ON table1.common_field = table2.common_field;
```

> Note: In some databases, RIGHT JOIN is called RIGHT OUTER JOIN.

Example

Let's say we want to return all orders and any customers that have placed an order:

The CUSTOMERS table contains information regarding the customers, while the ORDERS table contains information regarding orders placed by customers. As we want the information about all orders and any customers that have paced an order, so we will use RIGHT JOIN.

Customer Table

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	John	25	NY	65,000
3	Emily	23	Ohio	20,000
4	Bill	25	Chicago	75,000
5	Tom	27	Washington	35,000
6	Jane	22	Texas	45,000

Orders Table

Order_Id	Date	Customer_Id	Amount
100	2019-09-08	2	5000
101	2019-08-20	5	3000
102	2019-05-12	1	1000
103	2019-02-02	2	2000

Similar to INNER JOIN, if we want to use RIGHT JOIN we must specify a common column between the two tables.

Customer Table

ID	NAME	AGE	ADDRESS	SALARY
1	Mark	32	Texas	50,000
2	John	25	NY	65,000
3	Emily	23	Ohio	20,000
4	Bill	25	Chicago	75,000
5	Tom	27	Washington	35,000
6	Jane	22	Texas	45,000

Orders Table

Order_Id	Date	Customer_Id	Amount
100	2019-09-08	2	5000
101	2019-08-20	5	3000
102	2019-05-12	1	1000
103	2019-02-02	2	2000

Common column between the two tables

The RIGHT JOIN returns all of the records in the ORDERS table along with the matched reords in the CUSTOMERS table.

ID	NAME	AMOUNT	DATE
2	John	5000	2019-09-08
5	Tom	3000	2019-08-20
1	Mark	1000	2019-05-12
2	John	2000	2019-02-02

Like before we use SELECT to display the desired columns only.

3 of 3

The SQL query to retrieve all orders and some of the customers(those who have placed an order):

```
SELECT  CUSTOMERS.ID, CUSTOMERS.NAME, ORDERS.AMOUNT, ORDERS.DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

As you can see, the RIGHT JOIN keyword returns all records from the right table (ORDERS), even if there are no matches in the left table (CUSTOMERS).

Quick quiz!

Q

Will the following query return the NAME and ADDRESS of the customer

that ordered an item along with the items' ORDER_ID?

```
SELECT CUSTOMERS.NAME, CUSTOMERS.ADDRESS ,ORDERS.ORDER_ID
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON ID = CUSTOMER_ID;
```

Exercise 1

This first exercise will test your knowledge of aggregate functions.

WE'LL COVER THE FOLLOWING

Problem statement

Write a SQL query to fetch the count of employees working on project ‘P1’.

Input

The following tables are available to you as inputs:

Employee Table

Emp_Id	Full_Name	Joining_Date
100	John Smith	2019/02/20
101	Anthony Williams	2019/03/01
102	Ethan Carter	2019/01/28

Salary Table

Emp_Id	Project	Salary
100	P1	20,000
101	P2	40,000
102	P3	50,000
103	P3	50,000
104	P2	40,000

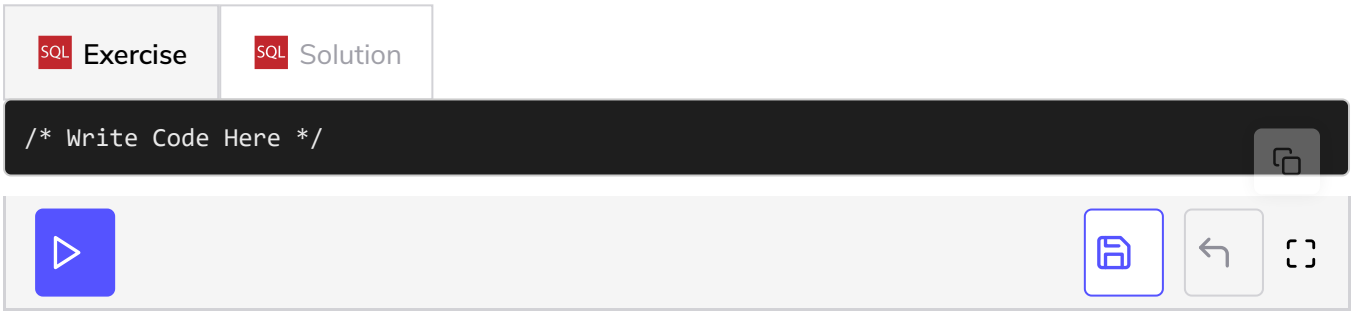
103	Mathew Mercer	2019/04/15
104	Nolan North	2019/05/19
105	Ashley Jenkins	2019/09/20
106	Amy Madison	2019/07/05
107	Emily Simpson	2019/08/22
108	Betty White	2019/07/09

105	P4	15,000
106	P1	20,000
107	P5	70,000
108	P5	55,000

Challenge

Write your query in the widget below. You don't need to worry about creating the tables as they have already been created for you. All you need to do is write the query mentioned in the problem statement.

If you feel stuck or think you have written the right query, the solution is discussed in the next lesson.



Solution to Exercise 1

Solution to exercise 1.

WE'LL COVER THE FOLLOWING

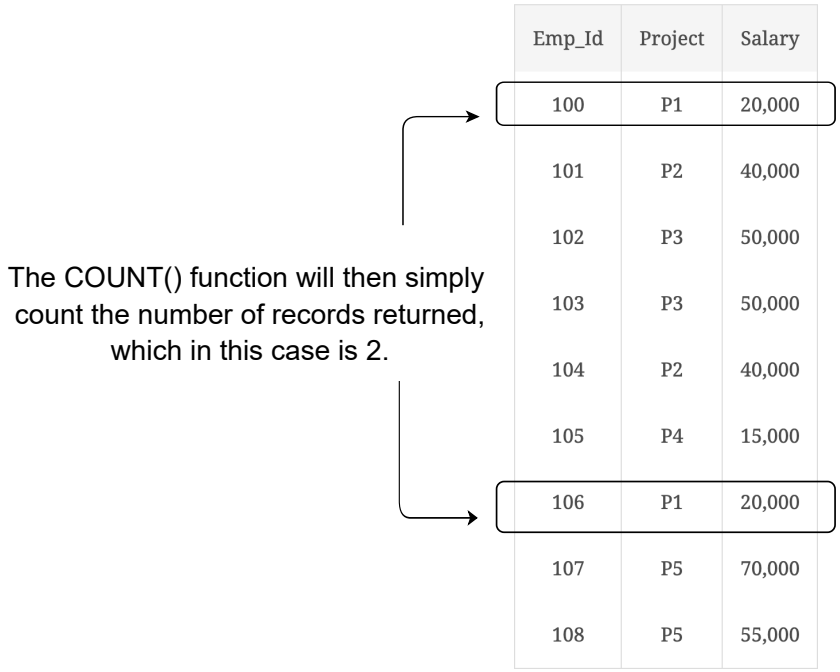
Solution

```
SELECT COUNT(EMP_ID)
FROM SALARY
WHERE PROJECT = 'P1';
```

Explanation

Nothing too tricky going on here. All we need to do is use the `COUNT()` function to get the number of employees and in the `WHERE` clause, specify the project we are interested in i.e., P1. The slides below provide a visual representation of the solution:

The WHERE clause will return those records where Project = P1



Emp_Id	Project	Salary
100	P1	20,000
101	P2	40,000
102	P3	50,000
103	P3	50,000
104	P2	40,000
105	P4	15,000
106	P1	20,000
107	P5	70,000
108	P5	55,000

Exercise 2

This first exercise will test your knowledge of JOINS.

WE'LL COVER THE FOLLOWING

Problem statement

Write a SQL query to fetch employee names having a salary greater than or equal to $40,000 and less than or equal $60,000.

Input

The following tables are available to you as inputs:

Employee Table

Emp_Id	Full_Name	Joining_Date
100	John Smith	2019/02/20
101	Anthony Williams	2019/03/01
102	Ethan	2019/01/

Salary Table

Emp_Id	Project	Salary
100	P1	20,000
101	P2	40,000
102	P3	50,000
103	P3	50,000

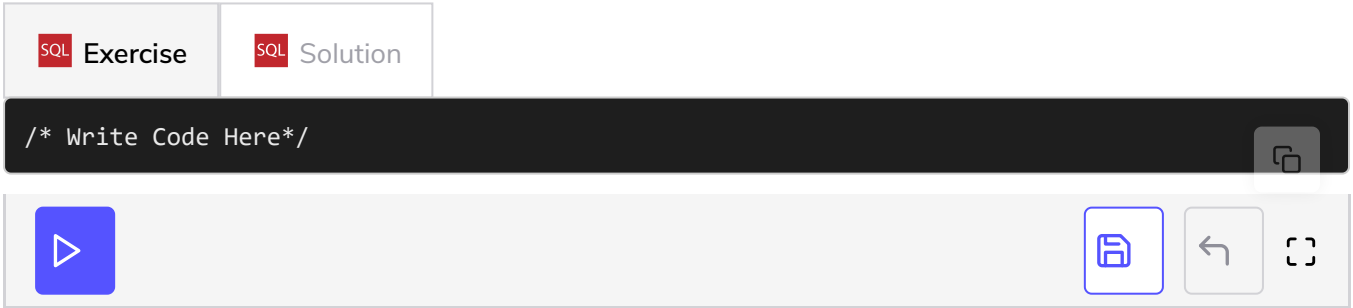
	Carter	28
103	Mathew Mercer	2019/04/15
104	Nolan North	2019/05/19
105	Ashley Jenkins	2019/09/20
106	Amy Madison	2019/07/05
107	Emily Simpson	2019/08/22
108	Betty White	2019/07/09

104	P2	40,000
105	P4	15,000
106	P1	20,000
107	P5	70,000
108	P5	55,000

Challenge

Write your query in the widget below. You don't need to worry about creating the tables as they have already been created for you. All you need to do is write the query mentioned in the problem statement.

If you feel stuck or think you have written the right query, the solution is discussed in the next lesson.



Solution to Exercise 2

Solution to exercise 2.

WE'LL COVER THE FOLLOWING

Solution #1

```
SELECT E.FULL_NAME, S.SALARY
FROM EMPLOYEE AS E
INNER JOIN SALARY AS S
ON E.EMP_ID = S.EMP_ID
WHERE S.SALARY >= 40000 AND S.SALARY <= 60000
```

The solution will also work without using the aliases as seen below:

Solution #2

```
SELECT EMPLOYEE.FULL_NAME, SALARY.SALARY
FROM EMPLOYEE
INNER JOIN SALARY
ON EMPLOYEE.EMP_ID = SALARY.EMP_ID
WHERE SALARY.SALARY >= 40000 AND SALARY.SALARY <= 60000
```

Explanation

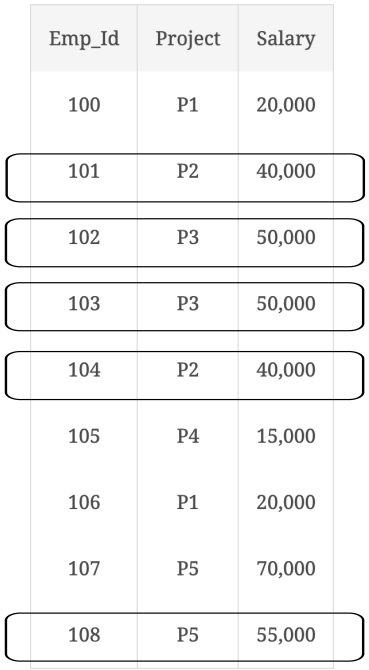
We will perform an INNER JOIN on the two tables to retrieve the common

records and then the WHERE clause is used to specify the condition that SALARY has to be in a certain range.

It is important to note that we joined the two tables because part of the information we needed was in the EMPLOYEE table (`Full_Name`) and the other half in the SALARY table (`Salary`).

The slides below help to visualize the solution:

The WHERE clause will return those records that fall in the specified salary range



Emp_Id	Project	Salary
100	P1	20,000
101	P2	40,000
102	P3	50,000
103	P3	50,000
104	P2	40,000
105	P4	15,000
106	P1	20,000
107	P5	70,000
108	P5	55,000

1 of 2

After the INNER JOIN combines the two tables, the SELECT statement will display the columns we are intrested in

FULL_NAME	SALARY
Anthony Williams	40000
Ethan Carter	50000
Mathew Mercer	50000
Nolan North	40000
Betty White	55000

Exercise 3

This exercise will test your knowledge of GROUP BY and ORDER BY.

WE'LL COVER THE FOLLOWING

Problem statement

Write a SQL query to fetch a project-wise count of employees sorted by the project's count in descending order.

Input

The following tables are available to you as inputs:

Employee Table

Emp_Id	Full_Name	Joining_Date
100	John Smith	2019/02/20
101	Anthony Williams	2019/03/01
102	Ethan Carter	2019/01/28

Salary Table

Emp_Id	Project	Salary
100	P1	20,000
101	P2	40,000
102	P3	50,000
103	P3	50,000
104	P2	40,000

103	Mathew Mercer	2019/04/15
104	Nolan North	2019/05/19
105	Ashley Jenkins	2019/09/20
106	Amy Madison	2019/07/05
107	Emily Simpson	2019/08/22
108	Betty White	2019/07/09

105	P4	15,000
106	P1	20,000
107	P5	70,000
108	P5	55,000

Challenge

Write your query in the widget below. You don't need to worry about creating the tables as they have already been created for you. All you need to do is write the query mentioned in the problem statement.

If you feel stuck or think you have written the right query, the solution is discussed in the next lesson.

Solution to Exercise 3

Solution to exercise 3.

WE'LL COVER THE FOLLOWING

Solution

```
SELECT PROJECT, COUNT(Emp_Id) AS Emp_Count
FROM SALARY
GROUP BY PROJECT
ORDER BY Emp_Count DESC;
```

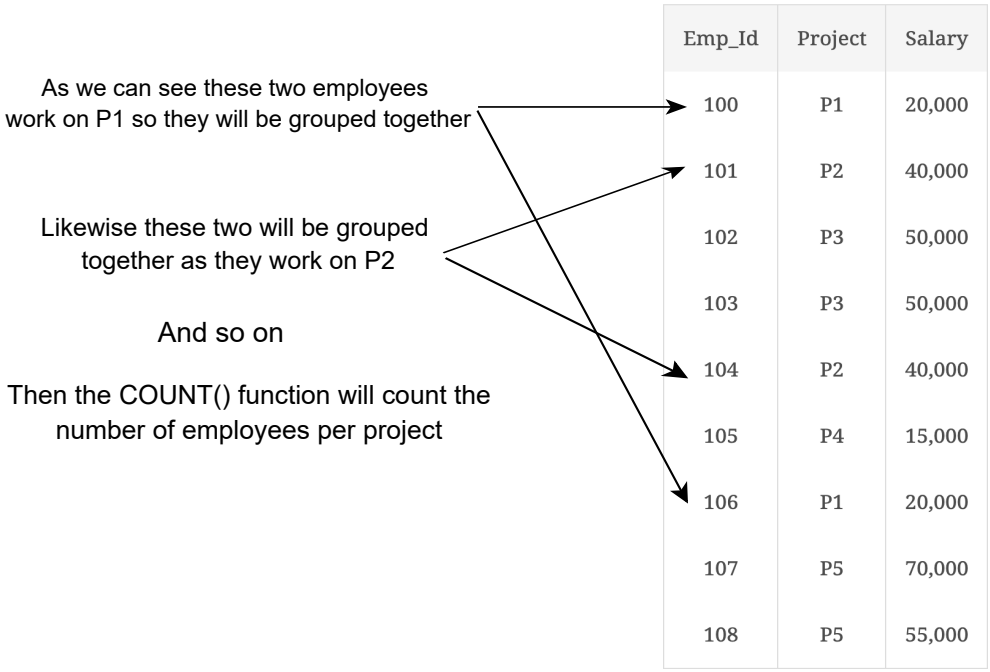
Again, the solution will work without using the alias for `COUNT(Emp_Id)`.

Explanation

The query has two requirements: first, fetch the project-wise count and then sort the result by that count. For a project-wise count, we will be using the COUNT() function to count the number of employees that have been grouped together using the GROUP BY clause. Lastly, for sorting, we will use the ORDER BY clause on the alias of the employee-count.

The slides below help to visualize the solution:

The GROUP BY clause will divide the employees working on same project into different groups.



Emp_Id	Project	Salary
100	P1	20,000
101	P2	40,000
102	P3	50,000
103	P3	50,000
104	P2	40,000
105	P4	15,000
106	P1	20,000
107	P5	70,000
108	P5	55,000

Then the ORDER BY clause will arrange the records in terms of the number of employees working on a project in descending order

PROJECT	Emp_Count
P1	2
P2	2
P3	2
P5	2
P4	1

Exercise 4

This exercise will test your knowledge of aggregate functions.

WE'LL COVER THE FOLLOWING

Problem statement

Write a SQL query to fetch the second-highest salary in the SALARY table.

HINT: You will need to nest one query inside another to solve this problem.

Input

The following tables are available to you as inputs:

Employee Table

Emp_Id	Full_Name	Joining_Date
100	John Smith	2019/02/20
101	Anthony Williams	2019/03/01
102	Ethan	2019/01/

Salary Table

Emp_Id	Project	Salary
100	P1	20,000
101	P2	40,000
102	P3	50,000
103	P3	50,000

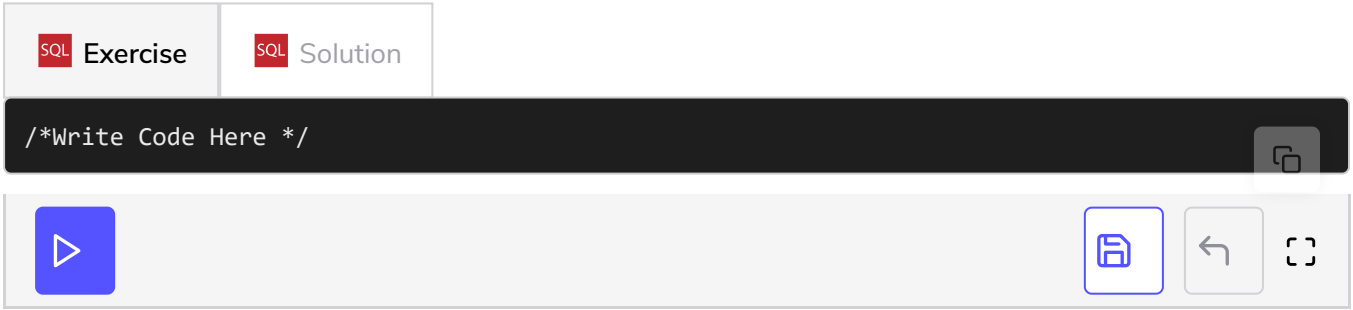
	Carter	28
103	Mathew Mercer	2019/04/15
104	Nolan North	2019/05/19
105	Ashley Jenkins	2019/09/20
106	Amy Madison	2019/07/05
107	Emily Simpson	2019/08/22
108	Betty White	2019/07/09

104	P2	40,000
105	P4	15,000
106	P1	20,000
107	P5	70,000
108	P5	55,000

Challenge

Write your query in the widget below. You don't need to worry about creating the tables as they have already been created for you. All you need to do is write the query mentioned in the problem statement.

If you feel stuck or think you have written the right query, the solution is discussed in the next lesson.



Solution to Exercise 4

Solution to exercise 4.

WE'LL COVER THE FOLLOWING

Solution

```
SELECT MAX(SALARY) AS SECOND_HIGHEST
FROM SALARY
WHERE SALARY < (SELECT MAX(SALARY) FROM SALARY);
```

Again, the solution will work without the `SECOND_HIGHEST` alias.

Explanation

The nested query in **line 3** will retrieve the highest salary from the table. Then the outer query will retrieve the second highest salary as the WHERE clause will return the salary that is below the maximum salary.

The slides below will help to visualize the solution:

The inner query will retrurn the record with the highest salary

Emp_Id	Project	Salary
100	P1	20,000
101	P2	40,000
102	P3	50,000
103	P3	50,000
104	P2	40,000
105	P4	15,000
106	P1	20,000
107	P5	70,000
108	P5	55,000

1 of 2

The outer query will then retrurn the record with the highest salary from the remaining records i.e. the second highest salary

Emp_Id	Project	Salary
100	P1	20,000
101	P2	40,000
102	P3	50,000
103	P3	50,000
104	P2	40,000
105	P4	15,000
106	P1	20,000
107	P5	70,000
108	P5	55,000

2 of 2

And with this last exercise, our course comes to an end. Before we part ways,

let's look at where we should go from here.

Where to Go from Here

In this lesson, we will conclude our course.

WE'LL COVER THE FOLLOWING

A Recap

We started the course by getting familiar with the history of databases and what came before databases caught on. We also looked at the reasons why we shifted towards databases.

We then moved on to the fundamentals of databases: characteristics, advantages, and a few examples.

Next, we looked at the basics of data modeling. We learned about the first step in database design, i.e., schemas. We also learned to classify databases subject to different attributes.

Next came entity-relationship diagrams. In this chapter, we discussed the different components that make up an ER diagram. Finally, we put together these concepts to make a university ER diagram.

At this point, we moved on to the fundamentals of relational databases. We familiarized ourselves with important ideas such as keys, constraints, and operations regarding relational databases.

The next topic that we covered was functional dependencies. Specifically, we focused on the rules and types of dependencies, as well as the representation of dependencies in our database schemas.

We then jumped into normalization. We studied different normal forms and

how to normalize a non-normalized table in order to reduce data redundancy.

Finally, we started our last chapter which was on Structured Query Language. We learned basic SQL syntax and then used those concepts to build queries in order to retrieve relevant data from the database.

Advanced Topics

This course was created to introduce you to the basic theory behind how databases work and how they are designed. Hence, there was only so much we could cover.

A couple of topics (namely enhanced entity-relationship diagrams, normal forms beyond BCNF, and advanced SQL queries) will be covered in the advanced course.

Thank you for sticking around until the end. We hope this course met your expectations and that you had fun learning about the fundamentals of database design. As always, keep practicing, keep having fun, and we hope to see you in another course soon.