INFORMATION SYSTEMS 1B
INSY6112/w
MODULE MANUAL 2025
(First Edition: 2012)

The Independent Institute of Education (Pty) Ltd is registered with the Department of Higher Education and Training as a private higher education institution under the Higher Education Act, 1997 (reg. no. 2007/HE07/002). Company registration number: 1987/004754/07.

DID YOU KNOW?

Student Portal

The full-service Student Portal provides you with access to your academic administrative information, including:

- an online calendar,
- timetable,
- academic results,
- module content,
- financial account, and so much more!

Module Guides or Module Manuals

When you log into the Student Portal, the 'Module Information' page displays the 'Module Purpose' and 'Textbook Information' including the online 'Module Guides or 'Module Manuals' and assignments for each module for which you are registered.

Supplementary Materials

For certain modules, electronic supplementary material is available to you via the 'Supplementary Module Material' link.

Module Discussion Forum

The 'Module Discussion Forum' may be used by your lecturer to discuss any topics with you related to any supplementary materials and activities such as ICE, etc.

To view, print and annotate these related PDF documents, download Adobe Reader at following link below:
https://www.adobe.com/acrobat/pdf-reader.html

IIE Library Online Databases

The following Library Online Databases are available. These links will prompt you for a username and password. Use the same username and password as for student portal. Please contact your librarian if you are unable to access any of these. Here are links to some of the databases:

Library Website	**This library website gives access to various online resources and study support guides** **[Link]**
LibraryConnect (OPAC)	**The Online Public Access Catalogue. Here you will be able to search for books that are available in all the IIE campus libraries.** **[Link]**
EBSCOhost	**This database contains full text online articles.** **[Link]**
EBSCO eBook Collection	**This database contains full text online eBooks.** **[Link]**
SABINET	**This database will provide you with books available in other libraries across South Africa.** **[Link]**
DOAJ	**DOAJ is an online directory that indexes and provides access to high quality, open access, peer-reviewed journals.** **[Link]**
DOAB	**Directory of open access books.** **[Link]**
IIESPACE	**The IIE open access research repository** **[Link]**
Emerald	**Emerald Insight** **[Link]**
HeinOnline	**Law database** **[Link]**
JutaStat	**Law database** **[Link]**

Table of Contents

Using this Manual

This manual has been developed to meet the specific objectives of the module and uses a number of different sources. It functions as a stand-alone resource for this module, and no prescribed textbook or material is therefore required. There may, however, be occasions when additional readings are also recommended to supplement the information provided. Where these are specified, please ensure that you engage with the reading as indicated.

Various activities and revision questions are included in the learning units of this manual. These are designed to help you engage with the subject matter as well as to help you prepare for your assessments.

Introduction

In our lives today, data is everywhere. We create more data every time we take a photograph or post something on social media. We shop online and read (or listen to) electronic books. We play games online, and we communicate with people around the world by means of electronic communication.

Data is the raw facts that get recorded; for example, a purchase that is made by a customer. That data can be processed into a format called information, which can be used for decision-making. (Coronel, et al., 2013) An online retailer can use statistical modelling to find out which products are most popular, for example. And that could inform decisions around which products to market most aggressively.

How does all this data get stored? How can we process it to create information that is relevant to businesses, governments, and individuals? How can we keep our data safe?

In this module, we will learn how relational databases work as well as NoSQL databases. And we will learn when it is most appropriate to use each of these technologies.

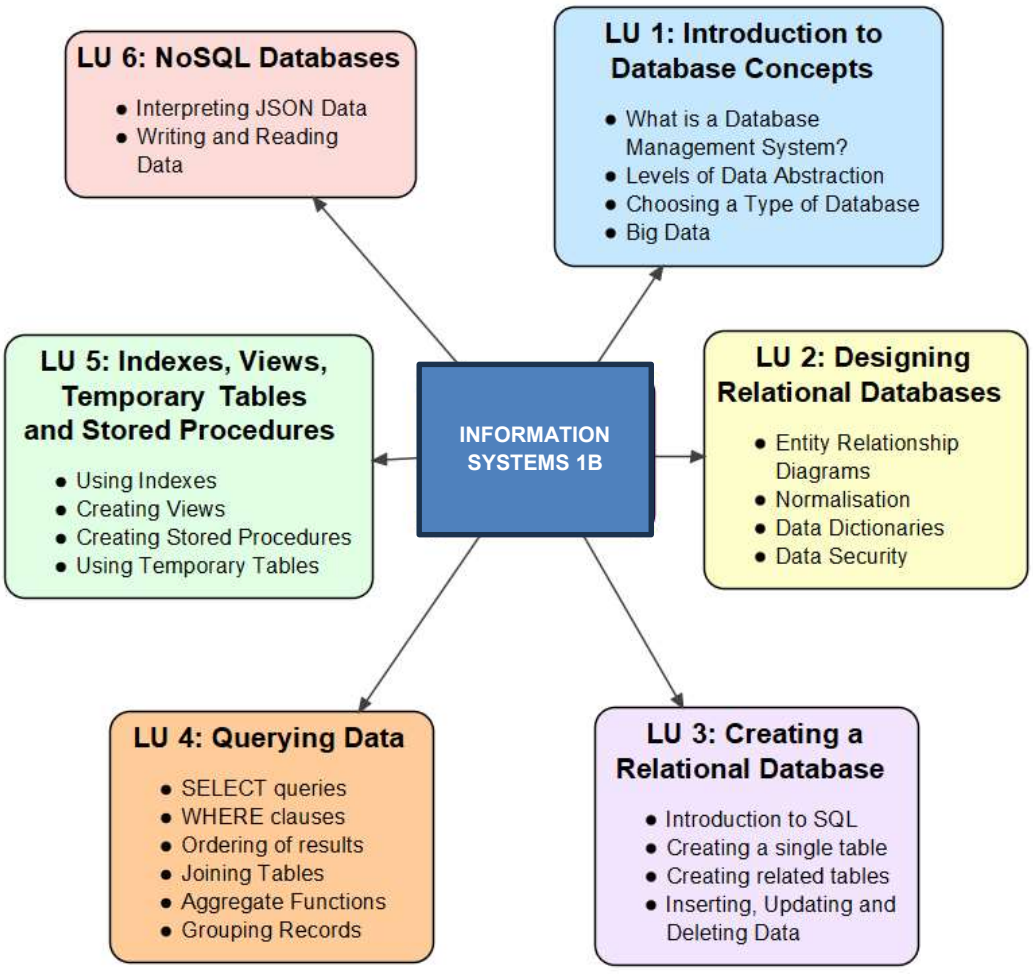


Figure 1. Module Outline

Glossary of Key Terms for this Module

Term	Definition	*My Notes*
Big Data	"High-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation." (Gartner, Inc., n.d.)	
Business Rule	"A brief, precise and unambiguous description of a policy, procedure or principle within a specific organisation." (Coronel, et al., 2013)	
Cardinality	"The maximum number of times an instance in one entity can relate to instances of another entity." (Lucid Software Inc., 2020)	
Data Abstraction	"The reduction of a particular body of data to a simplified representation of the whole." (Rouse, 2014)	
Database	"An organized collection of structured information, or data, typically stored electronically in a computer system." (Oracle, 2020)	
Data Lake	"Centralized repository that allows you to store all your structured and unstructured data at any scale." (Amazon Web Services, Inc., 2020)	
Declarative Languages	"Also called nonprocedural or very high level, are programming languages in which (ideally) a program specifies what is to be done rather than how to do it." (Encyclopædia Britannica, Inc., 2020)	
Entity	"A real-world object such as an employee or a project." (Watt & Eng, 2014)	

Term	Definition	*My Notes*
Entity Integrity	"Entity Integrity ensures that there are no duplicate records within the table and that the field that identifies each record within the table is unique and never null. "The existence of the Primary Key is the core of the entity integrity. If you define a primary key for each entity, they follow the entity integrity rule." (databasedev.co.uk, 2015)	
Entity Relationship Diagram	"A graphical representation of an organization's data storage requirements. Entity relationship diagrams are abstractions of the real world which simplify the problem to be solved while retaining its essential features." (Kirs, 2003)	
Foreign Key	"An attribute in a table that references the primary key in another table OR it can be null. Both foreign and primary keys must be of the same data type." (Watt & Eng, 2014)	
Functional Dependency	"A relationship between two attributes, typically between the PK and other non-key attributes within a table." (Watt & Eng, 2014)	
Index	"An on-disk structure associated with a table or view that speeds retrieval of rows from the table or view. An index contains keys built from one or more columns in the table or view." (Guyer, et al., 2019)	
JavaScript Object Notation (JSON)	"A lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate." (json.org, n.d.)	
Normalisation	"The process of determining how much redundancy exists in a table." (Watt & Eng, 2014)	

Term	Definition	*My Notes*
No-SQL Databases	“High-performance, non-relational data stores. They excel in their ease-of-use, scalability, resilience, and availability characteristics. Instead of joining tables of normalized data, NoSQL stores unstructured or semi-structured data, often in key-value pairs or JSON documents.” (Vettor, et al., 2020)	
Primary Key	“The column (field) in a relational database that uniquely identifies the row in the table. For example, account number is often a primary key. A ‘composite primary key’ or ‘super key’ is made up of two or more columns such as account number + name.” (PCMag Digital Group, 2020)	
Referential Integrity	“Referential integrity requires that a foreign key must have a matching primary key or it must be null. This constraint is specified between two tables (parent and child); it maintains the correspondence between rows in these tables. It means the reference from a row in one table to another table must be valid.” (Watt & Eng, 2014)	
Relationship	“An association among entities; for example, an employee works on many projects. A relationship exists between the employee and each project.” (Watt & Eng, 2014)	
Transaction	“MySQL transaction allows you to execute a set of MySQL operations to ensure that the database never contains the result of partial operations.” (mysqltutorial.org, n.d.)	
View	“A virtual table whose contents (columns and rows) are defined by a query.” (Milener, et al., 2020)	

Learning Unit 1: Introduction to Database Concepts	
Learning Objectives: • Explain what a database management system is. • Identify the different ways of categorising database management systems. • Identify the levels of data abstraction. • Identify the different types of databases. • Choose the appropriate type of database to use for a business problem. • Explain the concept of bBig dData.	**My notes**
Material used for this learning unit: • This module manual.	
How to prepare for this learning unit: • Read through the content in this learning unit.	

1 Introduction

Databases have been around for a long time. The first Database Management System (DBMS) was created all the way back in 1960! (Foote, 2020) Over the years, a huge amount has been written about databases. A Google Scholar search in July 2020 reveals that 28 800 articles with the word databases have been published just in the first half of 2020. (Google Scholar, 2020)

As a result, there is *a lot of theory* that we could go into. The focus of this module is more practical, though: how do we design and implement a database? So, this learning unit only introduces all the most important concepts that we need to know about to have a common vocabulary when we jump into the database design in Learning Unit 2.

2 Introduction to Databases

2.1 What is a Database Management System?

We already mentioned the term database management system (DBMS) in the introduction. But let us rewind a little and ask ourselves, what is a database?

Definition
"A **database** is an organized collection of structured information, or data, typically stored electronically in a computer system." (Oracle, 2020)

Let us look at an example of a system that makes use of data that is stored in a database – the system that tracks all the physical library books available in The IIE's libraries. In Figure 2, a search is shown that was looking for the recommended reading book for this module – the second item on the list. We see all the details of the book that has been captured on the system, that there are two copies in total, and that one of those are available.

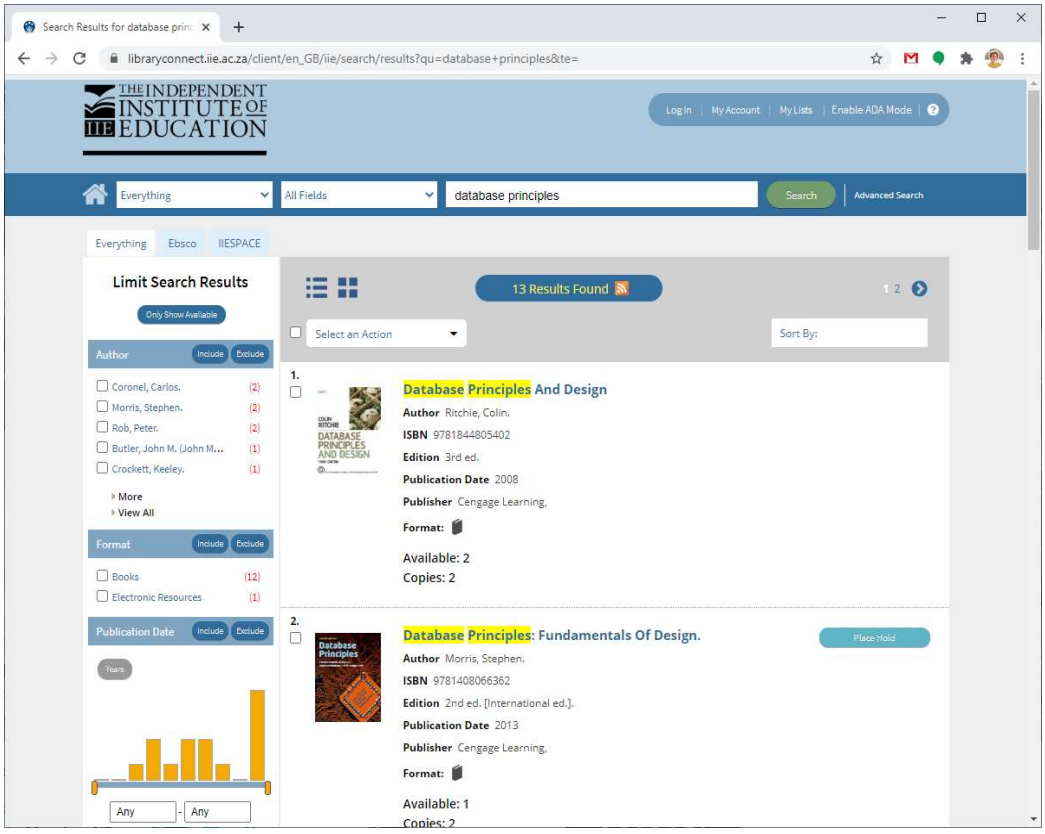


Figure 2. Search Results from The IIE Library (IIE Library, 2020)

How does this website have access to all this data? Well, it is stored in a central database somewhere. Every time a library user lends or returns a book, that is also recorded in the database.

The website that we are looking at is a user-friendly representation of the data. But there is a lot that is going on behind the scenes to make the system work. And it all starts with storing the data in the database and getting that data out again.

Historically, managing all that fell to the application developer to implement. We will look more at the history of data models in Section 3.1. But today we luckily have something that does most of the nitty-gritty details for us – it is called a database management system (DBMS).

Definition

"A **Database Management System** (**DBMS**) is software designed to store, retrieve, define, and manage data in a database." (OmniSci, Inc., 2019)

This definition has a lot of things that we are expecting. It is software that helps us, as software developers, make use of the database in our applications. In the library context, it allows us to add new books to the library database and retrieve the list of all the books with a certain keyword appearing somewhere in their data.

If you ask a software developer, something very close to this definition is likely what they will say about a DBMS. But a DBMS doesn't exist in isolation; it is part of a greater ecosystem in an organisation. For example, if you search online for the *components* that make up a DBMS, you will find at least the following items in addition to the software:

- **Hardware.** The DBMS software needs to run on servers, and the data needs to be stored somewhere. (Coronel, et al., 2013) In today's increasingly cloud-enabled world, those servers might not be physical boxes in a server room down the hall, but even cloud services still require hardware.
- **Data.** If you have a DBMS, but no data to store in it, it will serve no purpose. Data can be classified as either user data or meta data. The latter is "data about the data" – data that describes what user data will be stored, and in which data types. (Studytonight, 2020)
- **People.** There are lots of different types of people that make use of a database, from the administrators that should ensure smooth running of the DBMS software, to application developers that make use of the data in their apps, to end users that actually create and consume the data. (Coronel, et al., 2013)
- **Procedures.** Especially in larger companies, there will be rules around how the database is managed. (Data Entry Outsources, 2013) This is important since the data in a database is often vital to the operation of a company. Imagine for example, if somebody accidentally deleted all the products from an online retailer's database. The procedures that are put in place should prevent accidents like that.

A DMBS never exists in isolation. Whether it is an online retailer's database, the database that stores all the characters created in a massively multiplayer online (MMO) game, or the database that stores to-do items that you save locally on your phone,

there are always lots of different components involved in a system that makes use of a database.

2.2 Classification of Database Management Systems

In the very last paragraph of the previous subsection, we already got a small hint that there are different types of databases. The following information was adapted from (Watt & Eng, 2014, pp. 24-27), unless otherwise indicated.

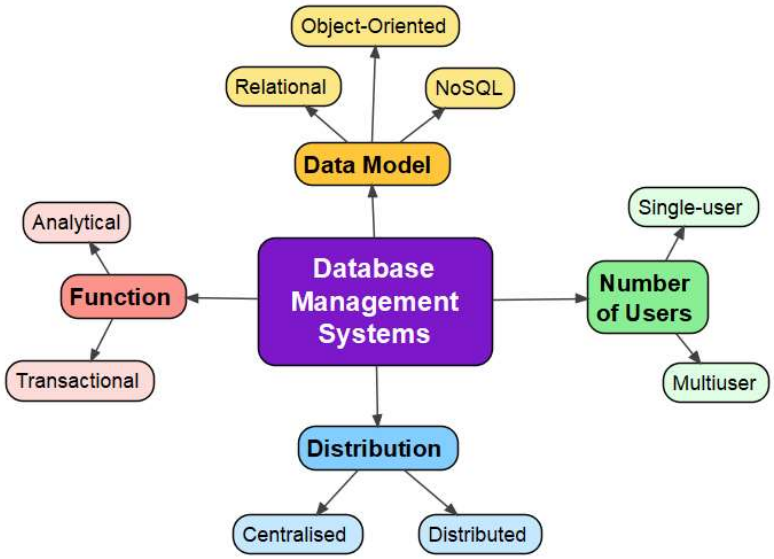


Figure 3. Classification of Database Management Systems

The first way in which databases can be classified is based on the **data model** that is used. Relational databases are still often used today, but there are also other, more recent, models. We will look at the different data models in more detail in Section 3.

The second way in which a database can be classified is by the **number of users** that it supports. A **single-user** database supports only one user at a time. For example, an SQLite database that is used by a to-do list app on an Android phone to store local data, is such a database. SQLite was designed from the ground up to be used in embedded and mobile applications like that. (SQLite.org, n.d.)

In contrast, a **multi-user** database is one that can be accessed by many users at the same time. For example, lots of players can access their MMO characters at the same time. And many shoppers can browse the catalogue of an online retailer at the same time. Systems like that need to be designed to support concurrency, starting with the DBMS that makes data access possible.

The third way of classifying a DBMS is according to its **distribution:** it can either be centralised or distributed. A **centralised** database is one where the database is stored in one central spot, from where it is accessed by other systems. A **distributed database** is one where the data is stored at multiple sites.

Distributed databases can be **homogenous** (using the same software everywhere) or **heterogenous** (using different software at different sites). If the software that is used is not the same, then there must be a mechanism to exchange data between the different sites.

The fourth way of classifying a DBMS is by its main **function:** whether it is operational (also called transactional) or analytical. (Simpson, 2016) A **transactional** system is one that supports the normal operations of an organisation. For example, an e-commerce site would store all the orders placed by a user in a transactional database. These databases focus on working with one piece of information at a time. (Simpson, 2016)

An **analytical database** has a different focus – it is used to make business decisions. The decisions are made on the same data that is generated during business operations but organised in a different way to make the queries faster or storing historical data. (Simpson, 2016) For example, the operators of a paid MMO might want to know which countries have the most players to focus marketing activities there. Additionally, how the number of players in each country changes over time could be used to measure the effectiveness of those marketing activities. So, the analytical database for that company should store data in a way that those queries over large datasets are optimised, rather than working with a single piece of information.

2.3 Levels of Data Abstraction

Before we consider the different levels of data abstraction, let's look at the concept of data abstraction.

Definition
"**Data abstraction** is the reduction of a particular body of data to a simplified representation of the whole." (Rouse, 2014)

When object-oriented software is designed, models can be created to communicate the design. These models are typically represented graphically, using the unified modelling language (UML). UML can be used to represent both the structural and behavioural aspects of a programme's design.

UML diagrams can function at different levels of abstraction. At the highest level, an architectural diagram would show large components that interact with one another. More detailed diagrams can then be created after that, all the way down to the class and method level.

2.3.1 A Software Development Life Cycle

Unless otherwise indicated, the following information was adapted from (Watt & Eng, 2014, pp.84-91).

Let's have a look at the process that is traditionally followed to develop software: the waterfall model. Although it is not used as often today, it still illustrates a lot of the basic concepts that we need to know about.

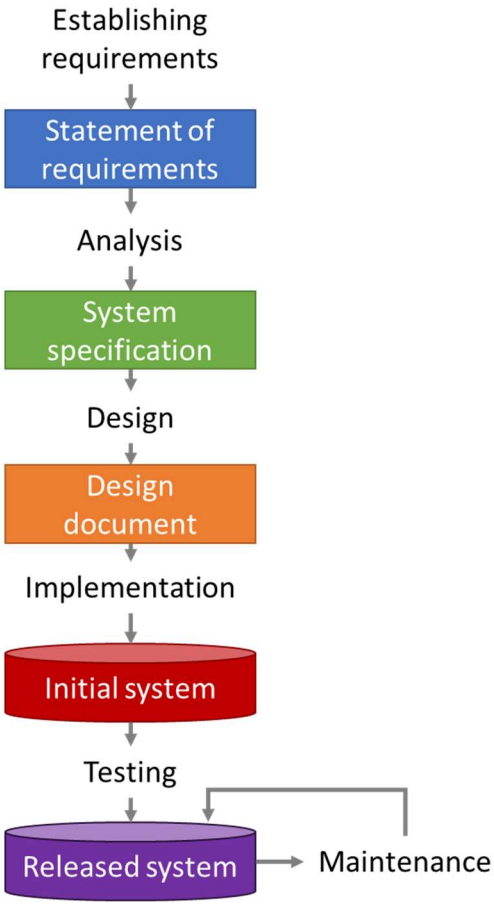


Figure 4. Waterfall Process (Watt & Eng, 2014)

In Figure 4, the shapes represent artifacts that are created during the software development life cycle (SDLC). And the steps outside the shapes are the activities that are conducted.

The process starts with **establishing the requirements**, talking to all the stakeholders, and finding out what is required. That is recorded in a statement of requirements (also sometimes called a user requirements specification).

Then **analysis** is performed to create a system specification. Here, we are still concerned with what the requirements are (i.e., what the system needs to do, not how). But now it is from the system's perspective.

The next phase is **designing** the system. Based on the system requirements, how should the system be constructed? This phase can include different levels of abstraction, from the architectural down to the class level, as previously discussed.

Next, the software is **implemented** according to the design. Then it is tested, released and finally enters a maintenance phase.

Software development today usually doesn't take place according to the waterfall model anymore – most teams opt for **agile methodologies**. Agile methodologies are iterative – all the steps are still followed in small increments. (Guru99, 2020) Specify a small piece of work, design just what is necessary, implement that one or two weeks' worth of work, test it, and deploy it. Then, specify the next small piece of work, and so on.

What does all of this have to do with databases? Well, there are also different levels of abstraction that get created during a process with multiple steps.

2.3.2 Levels of Data Abstraction

Just like software designs (especially UML diagrams) are used to communicate between team members working in a software team, we also want to create models of the data that we work with. Then all the stakeholders, with their different priorities and views on the data, can communicate effectively with one another by means of these models.

Figure 5 illustrates the different levels at which we can create models and how the models get more detailed as we get closer to implementation. There are other terms that are used by different authors to describe the various levels, but this figure shows the terms that will be used in this module manual.

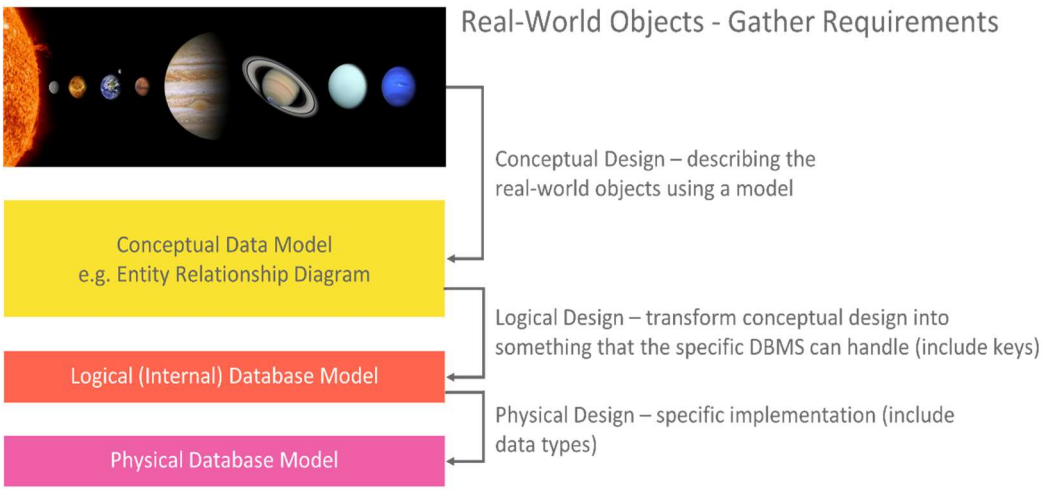


Figure 5. Database Design Process showing Levels of Data Abstraction (Steps from (Elgabry, 2016) and solar system image from (Comfreak, n.d.))

A **conceptual data model** is a model that describes the real-world objects (called entities) that we are working with, along with the relationships between those entities. This is a very high-level model that is still completely independent of the technology that we will eventually use to implement the database.

The **logical database model** contains the details that will be necessary to implement a specific kind of database; for example, a relational database would require us to add keys to the data model.

Finally, the **physical database model** includes things like the data types and how the data is physically stored by the DBMS.

So, as we go down the levels, the design gets more detailed, until we get to the actual implementation.

2.3.3 Database Life Cycle

The following information was adapted from (Watt & Eng, 2014, pp. 84-91), unless otherwise indicated.

How do these levels of abstraction fit into the bigger process of developing a database? The process is shown in Figure 6. The first step is again **establishing the requirements**. After talking to all the database users, the data requirements can be written down. This usually takes the form of business rules, which we will use a lot of in this module.

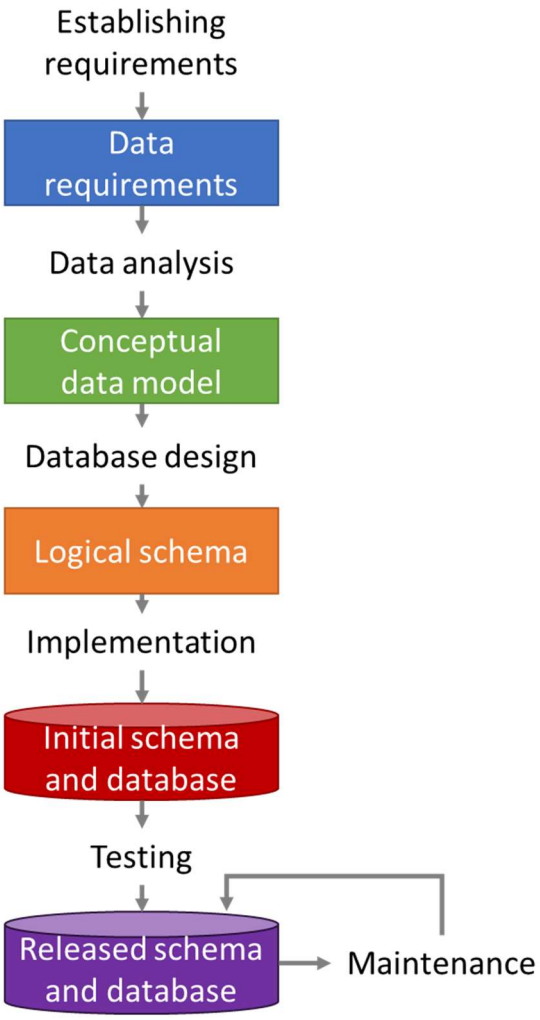


Figure 6. Waterfall Model for Database Development (Watt & Eng, 2014)

Definition

"A **business rule** is a brief, precise and unambiguous description of a policy, procedure or principle within a specific organisation." (Coronel, et al., 2013)

A business rule could be a rule for an actual business, for example: "A customer can place an order containing different quantities of multiple different stock items."

But the name 'business rule' can be misleading – it is, in fact, any description of the context in which the database operates. For example, in the astronomy example shown in Figure 5, a business rule could be: "A planet orbits around at least one star." Or in an MMO, a business rule could be: "A character can own from zero to many pieces of armour."

The next step is **data analysis,** where we take the requirements and transform them into a conceptual model. The conceptual model, just like the system requirements in

the software SDLC, must answer the question “What is required?” without getting into the how (implementation details).
The next step is **database design**, where we add the details that will be required for a specific type of database. For example, a relational database will require us to add keys to establish the relationships. And we choose the data types for the different fields.

Then we can **implement** the design. In the case of a relational database, the implementation is done using Structured Query Language (SQL). **Testing** may reveal issues that need fixing before the database is released. And finally, the database enters the **maintenance** phase.

In this module, we are going to look at how to create a conceptual model, how normalisation can be used to verify the conceptual designs, how to create a data dictionary that specifies the data types that we use. And then we will implement the database.

3 Types of Databases

So far, we have mentioned relational databases a couple of times. But that is not by any means the only type of database that is out there. Let us start by looking at how databases looked in history.

3.1 History of Data Models

Historically, data storage consisted of paper files stored in filing cabinets. This was a very time-consuming and laborious process. For example, a bank had to manually record every single transaction made by every client on paper. Living in today’s world, where electronic data storage is so pervasive, it is hard to imagine that systems could work like that.

Figure 7 (on page 21) shows a timeline of all the data models to date.

In the 1960s, electronic data storage became available. At first, data was stored in files that were basically just digital versions of the filing cabinets used before. (Coronel, et al., 2013) But after that, hierarchical and network models became available.

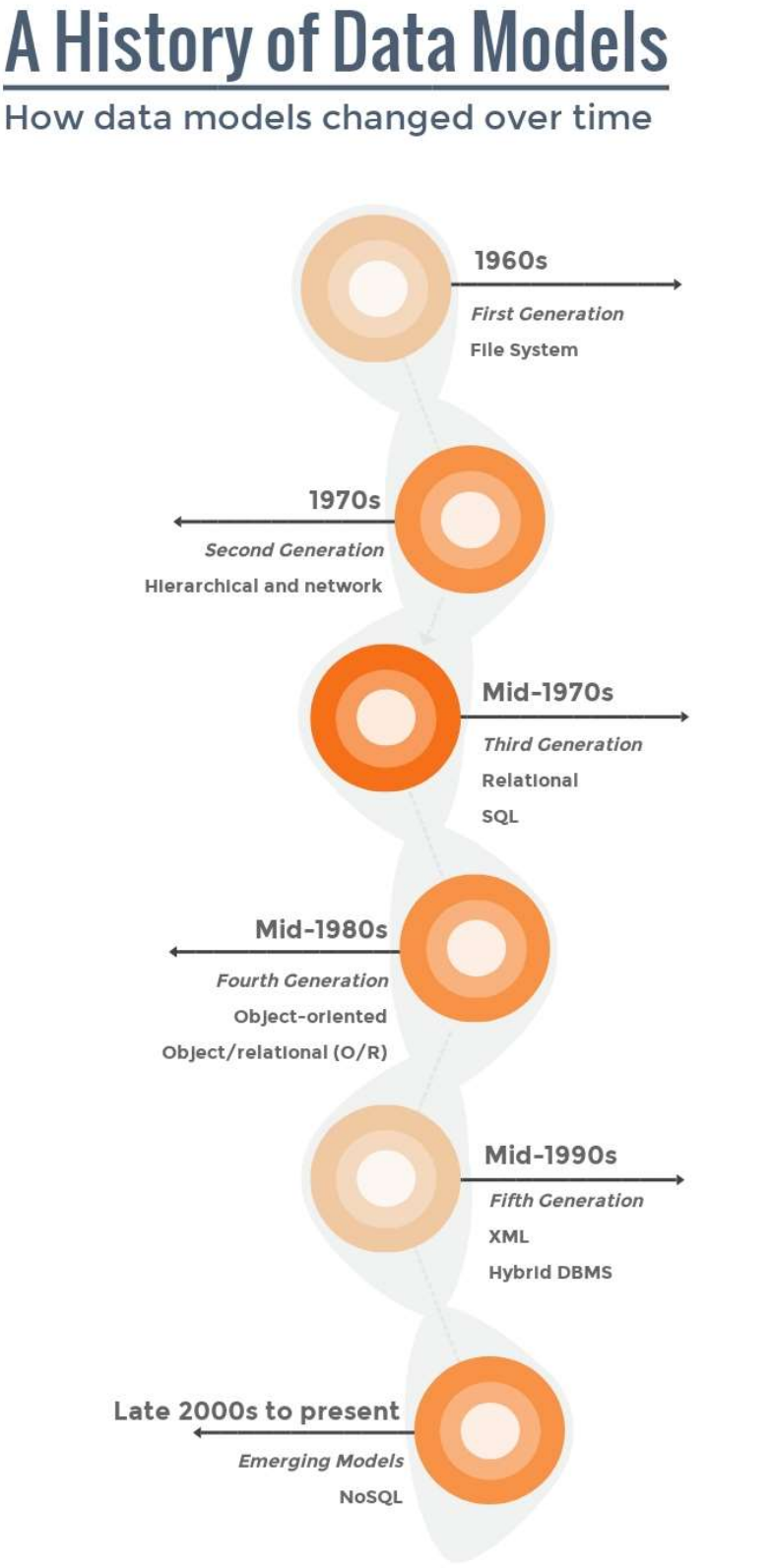


Figure 7. History of Data Models

In **hierarchical** models, records were stored in tree structures, which means that the data was restricted to one parent entity that related to multiple child entities (one-to-many relationships). (Foote, 2017)

Network models describe the data in a graph structure, which meant that they were more flexible in their representations of relationships. (Foote, 2017)

But if you have worked with databases at all, you will likely not have heard of either of these types of models before. That is because the **relational model**, first proposed in 1970, quickly became the model of choice for most developers. This model worked very well for **structured data**.

Object-oriented databases were introduced next, in the 1980s, and Extensible Markup Language (**XML**) based databases in the 1990s. The latest database developments are **NoSQL** databases. (Coronel, et al., 2013)

We will look in more detail at relational databases and NoSQL databases below. Those are the most frequently used data models today, and the focus of the rest of the module is in terms of implementation. But before we do that, let's look at one of the latest developments in data storage: a **data lake**.

Definition
"A **data lake** is a centralized repository that allows you to store all your structured and unstructured data at any scale." (Amazon Web Services, Inc., 2020)

A **data lake** is a unified way to store all the data that an organisation can possibly generate. Whether it is traditional, relational data, or files that have no structure, they can all be stored and processed in one place.

There are several different cloud providers that offer data lakes, and the exact features will vary. But most of those allow for advanced processing such as machine learning on these diverse types of data. And the data lakes are typically also designed to handle huge volumes of data. Microsoft Azure Data Lake for example, can store petabyte-size files. (Microsoft, 2020)

Read (Amazon Web Services, Inc., 2020) for a comparison of a data lake to a traditional data warehouse.

3.2 Relational Databases

The relational model was first proposed by Edgar F. Codd in 1970. (Foote, 2017) The paper that was the foundation for a large percentage of databases out there today, described "a model based on n-ary relationships, a normal form for database relations, and the concept of a universal data sublanguage." (Codd, 1970)

Definition

"**Declarative languages**, also called nonprocedural or very high level, are programming languages in which (ideally) a program specifies what is to be done rather than how to do it." (Encyclopædia Britannica, Inc., 2020)

The language that Codd proposed was declarative in nature. The idea of having a declarative language meant that developers could be much more productive. (Foote, 2017) They no longer needed to write the nitty gritty code that accessed the right spot in a file. Rather, they could use Structured Query Language (SQL) to specify what they want to get out of the database.

3.2.1 Tables

The most central concept in relational data models is the **table**. (Watt & Eng, 2014) (A table is also called a relation in this model, but to avoid confusion this module manual will not use that term.)

A **table** consists of **rows** and **columns**. Each row (also called a record) represents a set of related values. And each column is a specific attribute (also called a field). (Watt & Eng, 2014)

Table: Product

ID	Description	Unit Price	Category
1	Gaming Mouse USB 3200 DPI	149.00	Mouse
2	104 Key Cherry Red Keyboard	1 254.00	Keyboard
3	Ultra-Gaming Motherboard Mark III	2 799.00	Motherboard

Figure 8. Example of a Table: Product

Let us look at the example in Figure 8. Each row in this table represents a specific product in the catalogue of an online retailer. How do we know which product's unit price is highlighted in green value of 1 254.00? Well, it is related to the rest of the fields in that same row.

So, it is the unit price for the *104 Key Cherry Red Keyboard*. And what is the category? It is a *Keyboard*.

Is it possible for another product to also have the same price? Yes, it certainly is. If the product database is very large, it might even be quite likely that there are multiple products with the same price. So, in this small sample table, we could find the row with the specific price quite easily. But we need a way to uniquely identify a product in the database. Otherwise, how would the retailer know which product a customer wanted to buy?

In this table, there is a column called ID (short for identifier). The values in that column are unique – each row has a different ID. So, we can use that as the unique identifier. This is called the **Primary Key** (**PK**) of the table.

Definition

Primary Key: "The column (field) in a relational database that uniquely identifies the row in the table. For example, account number is often a primary key. A 'composite primary key' or 'super key' is made up of two or more columns such as account number + name." (PCMag Digital Group, 2020)

Each column in the table has a specific data type. The data types include number, character, date, and Boolean. (Watt & Eng, 2014)

Tables can be related to one another. Codd started his paper by saying that there will be "n-ary relationships" after all. So, let's look at relationships next.

3.2.2 Relationships

Relationships between data is a very central concept and we will discuss it in more detail later. There are fundamentally three types of relationships:

- **One-to-many relationship:** A row in one table is related to many rows in another table. For example, a customer can have many orders, but each order is made by exactly one customer.
- **Many-to-many relationship:** Many rows in one table are related to many rows in another table. For example, a customer can review many products, and the same product can be reviewed by many customers.
- **One-to-one relationship:** One row in a table is related to exactly one row in another table. For example, a department in the retailer has one manager, and the manager can only manage one department at a time.

How are relationships represented in tables? Using **Foreign Keys** (**FK**). An FK refers to the PK of the other (related) table. For example, for an order we would also store the PK of the customer that placed the order.

3.2.3 The Case for Relational Databases

We have already seen that relational databases have been around for a very long time. And yet it is still used very often today. Why is that?

"For organizations that need to store **predictable, structured data** with a finite number of individuals or applications accessing it, a relational database is still the best option. It offers a level of **maturity** and **widespread support** that remains unrivalled by current NoSQL alternatives, or any other alternatives for that matter." (Johnson, 2016)

When we think of data created in organisations, the nature of the data is often highly **structured**. We can know that a car that is assembled has a list of parts that make up the final car. And we can know exactly what type of information we are going to store about a part, like the part number, description, colour, type, etc. When we work with

customer data in online retail, we know that we will have a name, phone number, and address for each customer.
When we work with in-game data in an MMO, the kinds of data that we store is even more well known, since that is literally a world of our own creation.

NoSQL databases are much newer technologies. And often, people choose to use a technology simply because it sounds new and cool. Those databases do have their place – we will discuss that in the next subsection. However, the **choice of database** should be made based on the **nature of the data**.

If we have huge volumes of data, even millions of new rows created per day, relational databases can handle the load. If the **data** is inherently **structured**, then using a **relational database** is the best fit.

There are many different SQL databases available today. Some are open-source, like MySQL and PostgreSQL. Some are commercial products, like Microsoft SQL Server and Oracle Database. And many different options are available today to host relational databases in the cloud.

3.3 NoSQL Databases

> "**No-SQL databases** refer to high-performance, non-relational data stores. They excel in their ease-of-use, scalability, resilience, and availability characteristics. Instead of joining tables of normalized data, NoSQL stores unstructured or semi-structured data, often in key-value pairs or JSON documents." (Vettor, et al., 2020)

There are a lot of interesting things to note in this definition. Most importantly, NoSQL (short for Not Only SQL) databases store unstructured or semi-structured data.

Unstructured data is data that doesn't fit neatly into tables. For example, the contents of an email message have no specific structure and that means that it can't neatly be stored in table. Other examples include photos, files, and videos. (Marr, 2019)

Between structured and unstructured data, we find **semi-structured** data. And it is what it sounds like – it has properties of both structured and unstructured data. Consider again the example of an email message. The content is unstructured, but the header of the email contains information like the sender and recipients, which is structured in a known way. Looking at the whole email including the header, it is semi-structured data. (Marr, 2019)

If the nature of the data that you want to store is **semi-structured** or **unstructured**, then **NoSQL databases** are the best choice. The extra benefits that you get from using that then stem from the fact that NoSQL databases were designed from the ground up to handle the large-scale data that is created around the world today, known as big data. So, you can expect those databases to scale very well, and to have a very high availability.

Another feature is that NoSQL databases are considered easy to use. And while that is generally true (you don't need all the databases knowledge about relational databases to use those), each NoSQL database has its own way of accessing the data.

If you know SQL, you can use pretty much any relational database. But each NoSQL has something new that you need to learn.

3.3.1 Types of NoSQL Databases

There are four different types of NoSQL databases (Vettor, et al., 2020):

- **Document store:** Data is stored in documents, typically based on JavaScript Object Notation (JSON). MongoDB, the NoSQL database that we will use in the last learning unit of this module, is a document store.
- **Key-value store:** Data is stored as key-value pairs. Think of the way that a Dictionary in C# or a HashMap in Java stores data.
- **Column store:** Data is stored in rows and columns but is optimised for access in terms of the columns. (Fowler, n.d.)
- **Graph store:** Data is stored like in the graph data structure, with nodes, edges, and properties.

3.4 Big Data

The last concept that we need to explore before we conclude this introductory learning unit, is **big data**.

> "**Big data** is high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation." (Gartner, Inc., n.d.)

3.4.1 Three Vs of Big Data

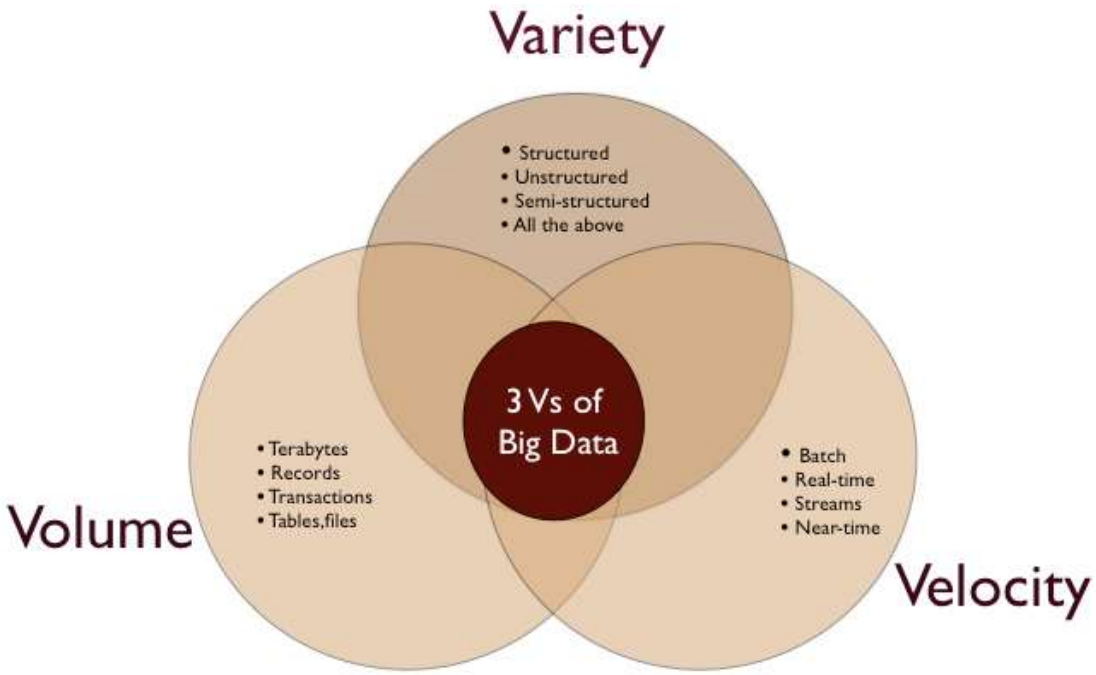


Figure 9. Three Vs of Big Data (Hijleh, 2016)

This definition introduces what is known as the three Vs of big data: volume, variety and velocity.

When it comes to **volume**, just imagine for a moment how many digital photos are taken around the world every day. It is estimated that more than 1.4 trillion (1,436,300,000,000) digital photos will be taken in the year 2020, which is a growth of 8.3% over 2019. (Carrington, 2020)

That leads us to **velocity** – data is getting created faster and faster, partially because a lot of data is getting generated automatically. (Oracle, Inc., 2020b)

We have already seen that there are three types of data: structured, semi-structured, and unstructured. And the sheer **variety** of different kinds of data (especially unstructured) is characteristic of big data.

Going beyond the original three Vs, other terms have been added too. For example, **value** and **veracity** (whether the data can be trusted). (Oracle, Inc., 2020b)

3.4.2 Sources of Big Data

Sources of big data include (Joshi, 2017):

- Media, including social media;
- Cloud data;
- Web pages;
- Internet of Things (IoT); and
- Traditional databases.

3.4.3 Using Big Data

Organisations have come to realise that big data holds a lot of potential value. So, what needs to happen to be able to get that value out of the data?

The first step is to **integrate** the **various sources** of data. (Oracle, Inc., 2020b) A company might for example, decide to integrate social media posts about their products with their existing transactional data, to get better insight into which products are successful.

The second step is to **store** the data somewhere, often in the cloud. (Oracle, Inc., 2020b)

The final step is to **analyse** the data – this is where the value is generated. Without analysis of the integrated data, it is worthless. (Oracle, Inc., 2020b)

4 Recommended Additional Reading

Amazon Web Services, Inc., 2020. *What is a data lake?.* [Online] Available at: https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/ [Accessed 11 December 2023].

Coronel, C., Morris, S., Crockett, K. and Rob, P. 2013. Chapters 1 and 2 in *Database Principles: Fundamentals of Design, Implementation and Management*. 2nd edition. Cengage Learning EMEA.

Vettor, R. et al., 2020. *Relational vs. NoSQL data.* [Online] Available at: https://docs.microsoft.com/en-us/dotnet/ architecture/cloud-native/relational-vs-nosql-data [Accessed 11 December 2023].

Watt, A. and N. Eng. 2014. Chapters 6 and 7 in *Database Design.* 2nd edition. Victoria, B.C. Available at: https://open.bccampus.ca/ browse-our-collection/find-open-textbooks/?uuid=5b6f010a-0563-44d4-94c5-67caa515d2c5 [Accessed 11 December 2023].

5 Revision Exercises

5.1 Revision Exercise 1

When is it more appropriate to use each of the following types of databases?

- Relational database
- NoSQL database

5.2 Revision Exercise 2

What are the different ways in which databases can be classified? Provide an example in each case.

6 Solutions to Revision Exercises

6.1 Revision Exercise 1

Read Sections 3.2 and 3.3 for the answer.

6.2 Revision Exercise 2

Read Section 2.2 for the theory behind this answer.

Learning Unit 2: Designing Relational Databases

Learning Objectives:	**My notes**
• Interpret an entity-relationship diagram. • Create entity relationship diagrams using Unified Modelling Language notation. • Review and revise an entity relationship diagram. • Distinguish between primary and foreign keys. • Identify the benefits of normalisation. • Normalise data to first normal form. • Normalise data to second normal form. • Normalise data to third normal form. • Explain what a data dictionary is. • Create a data dictionary. • Identify data that requires additional security.	
Material used for this learning unit: • This module manual. • Online diagramming tool: https://app.diagrams.net/ [Accessed 11 December 2023]. • ERDs for Game Developers using UML (see section 7 on p.82 for more information)	
How to prepare for this learning unit: • Read through the content in this learning unit.	

1 Introduction

In Learning Unit 1, we have explored some of the most important theoretical aspects of databases. Now we are ready to jump into designing a database. First, we are going to look at how to transform business rules into an entity relationship diagram (ERD) – a visual representation of the database design down to the logical level of abstraction. Then we will look at how normalisation of sample data can be used to improve the design, and how a data dictionary can be used to document the data types for each column. Lastly, we will look at security considerations when we design a database.

Before we jump into the details, there are two terms that we now need good definitions for: **entity** and **relationship**. These are the two most fundamental building blocks of database design.

Definition

"An **entity** represents a real-world object such as an employee or a project." (Watt & Eng, 2014)

When we were discussing the term business rule in Learning Unit 1, we mentioned that it is a term that goes beyond just business. Similarly, an entity is not necessarily just a real-world object – it can also be a game-world object or a purely digital construct like an account. It is anything that would be a **noun** in a **business rule**, that you can store data about in a database.

Definition

"A **relationship** represents an association among entities; for example, an employee works on many projects. A relationship exists between the employee and each project." (Watt & Eng, 2014)

Reading this definition, you might think that the example sounds a lot like a business rule. And that is exactly what it is. The **relationship** is the **verb** from the business rule – "works on" in the given example.

2 Entity Relationship Diagrams

2.1 Notations

When we looked at the levels of data abstraction in Learning Unit 1, we saw that software development models are often represented visually. And designing databases is no exception – we want to draw a picture that can clearly communicate what the structure of the database is going to be. This graphical representation is called an **entity relationship diagram** (**ERD**). It is, unsurprisingly, a diagram that shows the entities and their relationships.

Definition

"An **Entity Relationship Diagram** is a graphical representation of an organization's data storage requirements. Entity relationship diagrams are abstractions of the real world which simplify the problem to be solved while retaining its essential features." (Kirs, 2003)

Even though this definition dates all the way back to 2003, it is just as relevant even today. In fact, the idea of an ERD was proposed by Peter Chen in the 1970s. (Guru99, 2020b) And the **concepts** behind that original proposal are still used today.

Over the years, several different **notations** have been used to represent ERDs. What is a notation?

Definition

"A system of symbols used to represent special things." (MathsIsFun.com, 2018)

An example of a notation is musical notation, where the notes that are played are represented in a specific way. (MathsIsFun.com, 2018) Figure 10 shows an example of what this looks like.

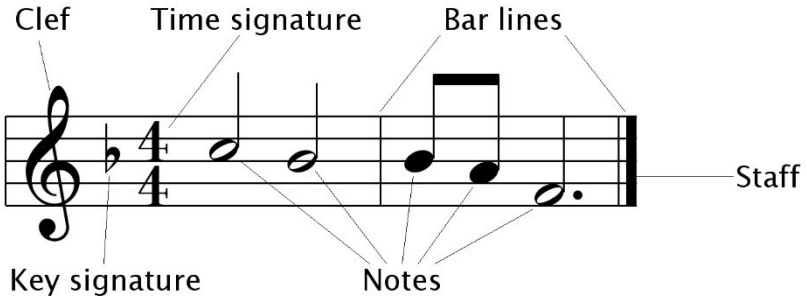


Figure 10. Musical Notation (Gillingham, 2007)

If you know how to read the musical notation, you will probably remember that it took time to learn how to do that.

If you don't understand this diagram at all, it makes the point that using a notation takes knowledge and practice. If I tell you that the higher the note is drawn on the bar, the higher the note would be when played, you now already have some idea of what the notation means. But there is still a lot more to learn.

The same is true for ERD notations. The diagrams were designed to be easy to understand on a high level. With some basic knowledge, you can get started with them. All the different notations have some features in common. But beyond the basics, there are lots of details that can be added.

The notations that have been used for ERDs include (Dybka, 2014):

- Barker's Notation
- Chen Notation
- IDEF1X Notation
- Arrow Notation
- UML Notation
- Crow's Foot Notation

If you do a web search about ERDs, you will get answers that seem comprehensive and good and look quite different from one another. And that is because these sources use different notations without mentioning which one is used.

In this module, we are going to draw our diagrams in UML. UML is a unified notation that can be used to model many different aspects of software, making it a very powerful visual language to know. But it is useful to be aware of the other notations, so that you can at least read those diagrams. The most common notations besides UML in use online (and in the workplace) is Crow's Foot and Chen.

2.1.1 Chen's Notation

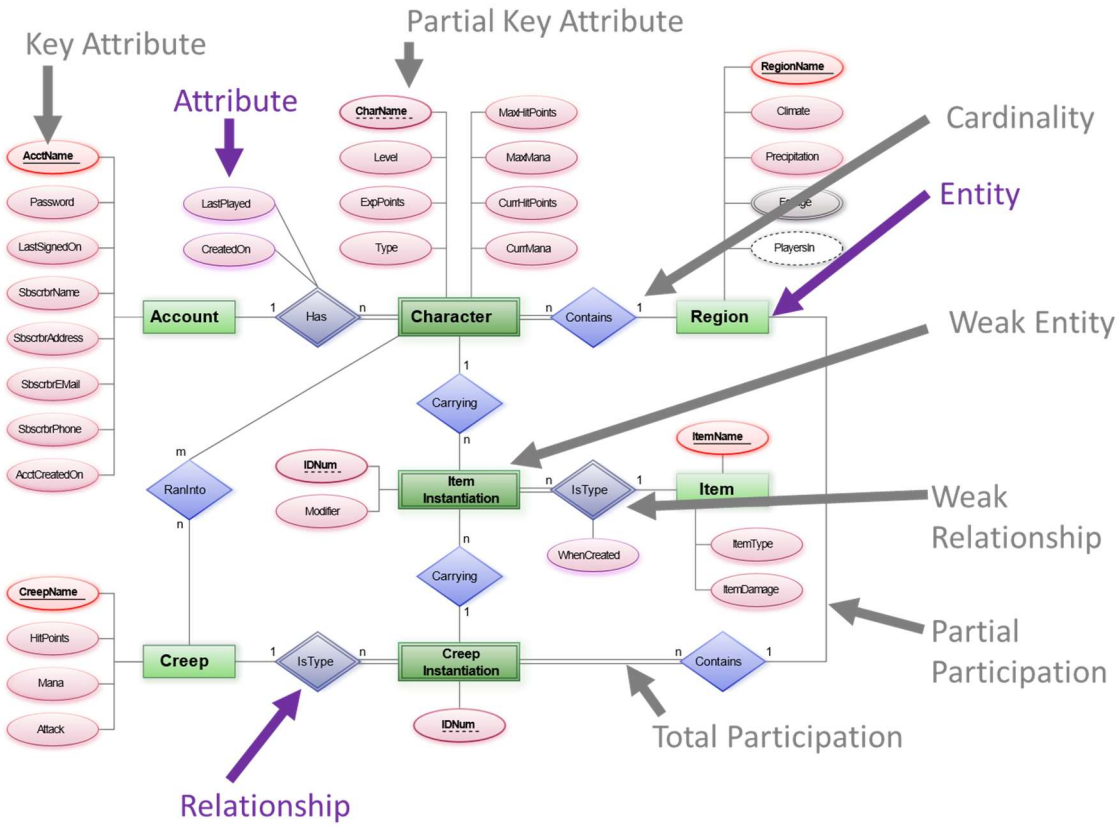


Figure 11. Chen's Notation (diagram from (Jorrit, 2011) with annotations)

Figure 11 shows an example of quite a complex model drawn in Chen's notation. The diagram shows the data model for an MMO, including aspects from both the accounting perspective of storing the account holder's details (top left), and the game perspective of storing characters and so on.

The annotations on the diagram point out all the interesting notational elements of Chen's notation, and as you can see, there are quite a lot of things. Let's stick to the basics here – the details won't make sense just yet. The three most important elements are shown in purple on the diagram:

- An **entity** is represented as a **rectangle** with the name of the entity.
- A **relationship** is represented as a **diamond** shape, again with the name of the relationship inside it. It is connected to the entities that are part of the relationship with lines.
- An **attribute** is represented as an **ellipse**, linked with a line to the entity (or even relationship) that the attribute belongs to.

If you want to distinguish between the different notations, the giveaway for Chen's notation is the diamond shape that represents the relationship.

Read (Dybka, 2014b) for more details about drawing a diagram using Chen's Notation. Also read Chapter 8 in (Watt & Eng, 2014).

2.1.2 Crow's Foot Notation

Crow's Foot notation was also first proposed in the 1970s. (Dybka, 2016) It is also known, less commonly, as Information Engineering (IE) notation. (Object Management Group, n.d.)

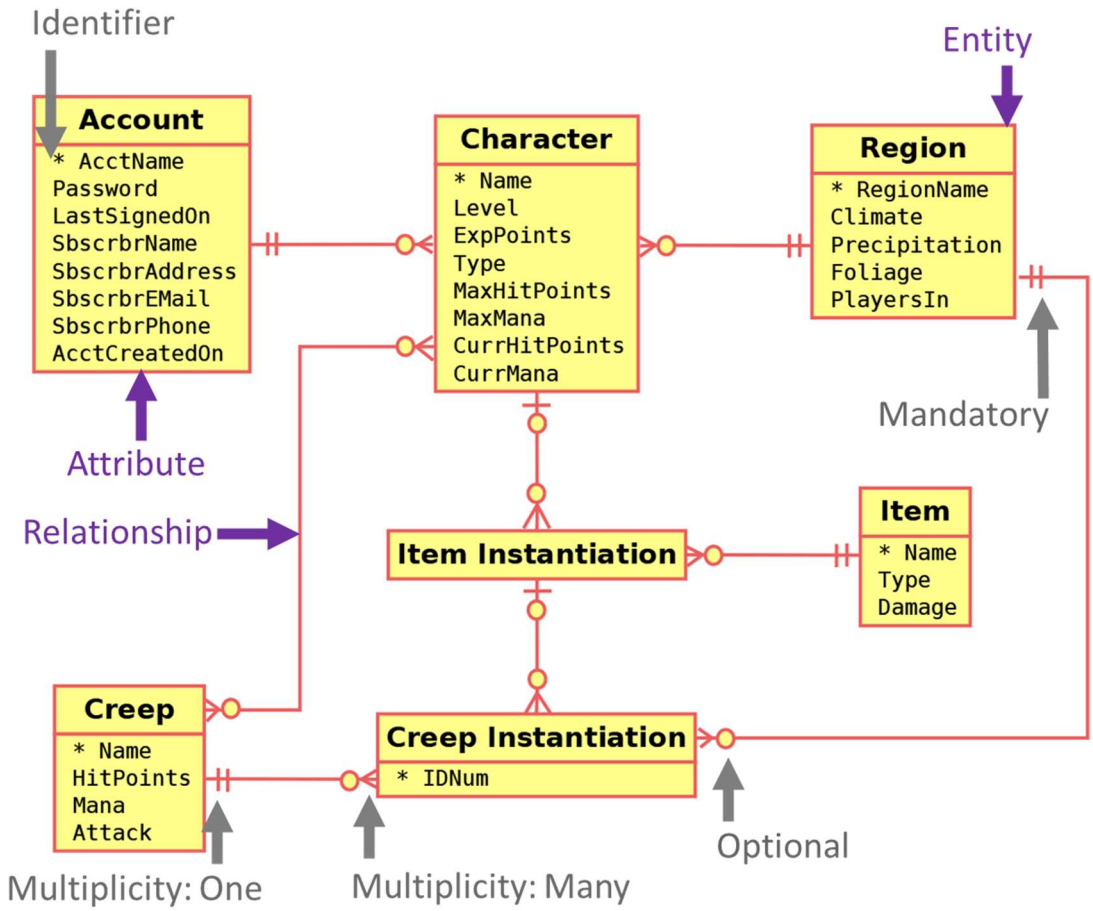


Figure 12. Crow's Foot notation (based on the model from Figure 11)

The diagram in Figure 12 represents the same model as the one in Figure 11, but this time in Crow's Foot notation. The most important aspects to take note of here are:

- An **entity** is a **rectangle** with the name of the entity at the top. And below that, in the same rectangle, are the **attributes** of the entity. This diagram looks simpler because of this more compact representation.
- A **relationship** is a line that connects the two entities.

In terms of the entity notation, there are subtle differences in exactly how different implementations draw the information. Have a look at (Garcia, 2019) for example, where screenshots from four different tools are shown.

The distinguishing feature of a Crow's Foot diagram is the way that the multiplicities are drawn. The multiplicity of many has the shape of a crow's toes.
Read (Dybka, 2016) for more details about drawing a diagram using Crow's Foot notation.

2.1.3 Unified Modelling Language (UML) Notation

The remainder of this section is going to be spent looking at ERDs in UML notation in detail. But for completeness, let's have a quick look at a basic diagram for comparison. Figure 13 shows part of the same model in UML notation. The important parts are:

- An **entity** is represented as a **rectangle**, with the name at the top and the **attributes** below that.
- A **relationship** is a **line** between the two entities, with numbers representing the minimum and maximum number of entities that are involved in the relationship.

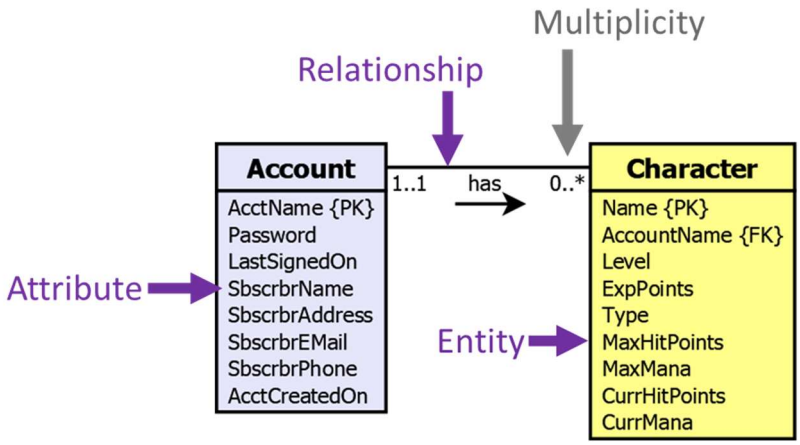


Figure 13. UML Notation (based on Figure 11)

There is no standard way to show keys in UML notation. But the notation described by (Coronel, et al., 2013) uses a postfix {PK} and {FK}, and that works really well. We will stick to that convention in this module.

The way in which the numbers are written is the most distinguishing feature of a UML diagram.

Read (Dybka, 2014c) for more details about drawing a diagram using UML notation. Also read Chapter 5 of (Coronel, et al., 2013) for very detailed information about the various aspects of ERDs plus two great case studies.

2.2 Process for Developing ERDs

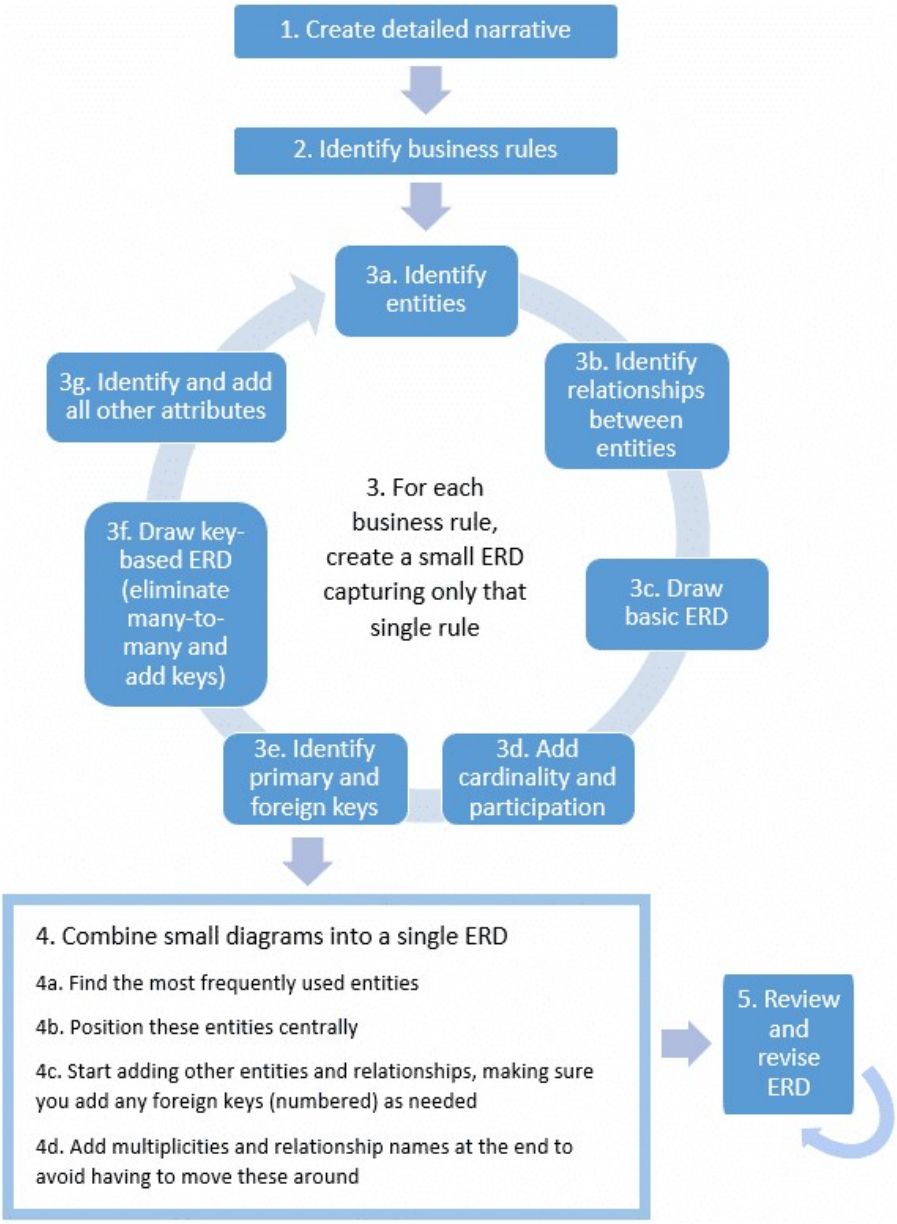


Figure 14. ERD Development Process (Pellissier, 2019)

A step-by-step process for creating an ERD has been described by several sources including (Dalbey, 2005). The process involves identifying the entities and relationships and adding the details around multiplicities and attributes. If you follow the steps, you will end up with a good diagram.

Two steps were added to the beginning of the process by (Coronel, et al., 2013):

1. Create detailed narrative; and
2. Identify business rules.

These two steps are essential in a real business case, where the system that is built must meet a real business need. For the purposes of this module, though, we will assume that we have the business rules already.

We are going to take the iterative approach shown in Figure 14. Do all the steps for each business rule, creating a small ERD, and then we combine those small diagrams afterwards into a bigger picture.

For the purposes of our discussion, we are going to work with business rules around **Formula 1 racing teams**. We will be designing a database that stores the data around teams, drivers, and races.

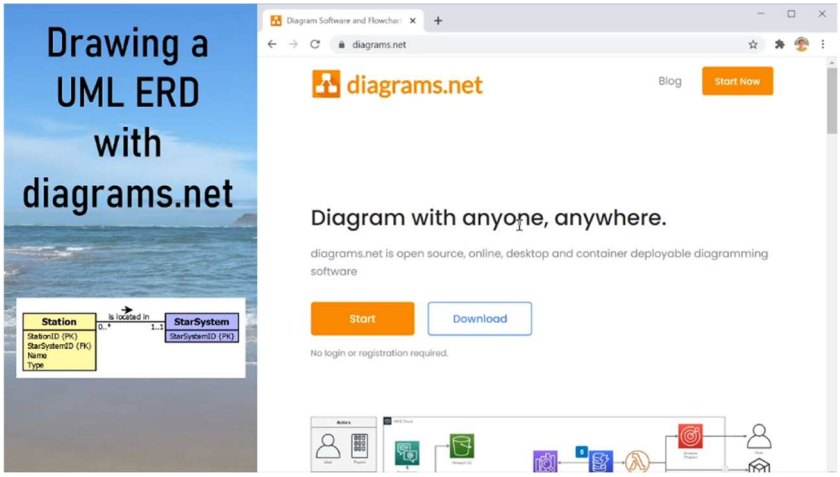
Figure 15. Formula 1 Racing (Fischer, n.d.)

The information presented here is not endorsed in any way by Formula One World Championship Limited, but if you are interested in learning more from the official source, visit their website at: https://www.formula1.com/en.html [Accessed 11 December 2023].

Any tool that can draw **UML class diagrams** can be used for drawing ERDs in UML notation. The recommendation for this module is using https://app.diagrams.net/ [Accessed 11 December 2023]. previously known as draw.io). This is the link to the website, where you can create lots of different diagrams completely for free.

But there is also a desktop app that you can download from that same site if you want to use it directly on your own computer.

Start by creating a **blank diagram**, and then use the shapes under the UML category. Don't start with the ERD template – that uses Crow's Foot notation!



Watch this YouTube video that explains how to use diagrams.net to draw a UML ERD: https://youtu.be/miGNDlDNInM [Accessed 11 December 2023].

2.3 Entities

Business Rule 1

When a team enters, the following information needs to be provided: team name, location of where the team is based, the power unit used by the team and the team colours.

Step 3a in Figure 14 for drawing a diagram for a specific rule is to identify the entity or entities that are involved. Entities can be identified by looking at the nouns in the business rule.

Definition

"A **noun** is a word that refers to a thing (book), a person (Betty Crocker), an animal (cat), a place (Omaha), a quality (softness), an idea (justice), or an action (yodeling). It's usually a single word, but not always: cake, shoes, school bus, and time and a half are all nouns." (Merriam-Webster, n.d.)

Which nouns can we identify here? **Team**.

This is a very simple business rule that just establishes that **Team** will be an entity, and a few attributes of the team.

Let us start by just creating the entity called Team using diagrams.net.

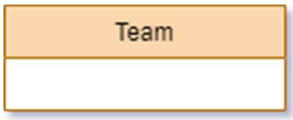


Figure 16. Business Rule 1 – Team Entity

There are a few interesting remarks that we can make about this very simple diagram already. We see the rectangular shape that we are expecting, with the name of the entity. The default colour scheme of diagrams.net is black and white. But it can be useful to colour-code the entities if that is not the only way to convey information. Always keep in mind that colour-blind people might read your diagrams or use your software!

2.3.1 Naming Entities

When naming things on your diagrams, always be **consistent**. Do not use different words (synonyms) for the same concept (like store and shop) – that will just cause confusion. And do avoid homonyms too (words that sound the same but have different meanings).

The **name** of the entity is **singular**. Here, the name of the entity is **Team**, not Teams. This is done because it makes more logical sense in a lot of different cases. If you are going to create a class in a programming language like Java or C# to represent this data, what would you call it? Well, it stores the data and has the behaviours related to a specific Team. So, it would be called Team, and if a new instance is created, it would look like this:

```
Team team = new Team();
```

Would this same line of code make sense if you read it out loud if the class were called Teams? No, it sounds like it should be managing more than one team, and it isn't. You will see that the same kind of thing also applies when writing queries in SQL. So, to keep everything consistent across the database and the programs that make use of the database, stick to the same naming convention: **singular names**.

2.4 Attributes

There are no relationships in this first rule, so we can skip most of the steps in Figure 14. But we do need to add some attributes (step 3g). So, let us skip ahead to that step.

Our business rule specified that the **Team** entity should have a few attributes. Before we add those to the diagram, let's look at the different types of attributes that can exist.

Single-valued attributes (also called simple attributes) can only have a single value. (Watt & Eng, 2014) In the team example, the name of the team is a single-valued attribute. One of the real teams competing in 2020 is called "Alfa Romeo Racing". Even though this name consists of multiple words, you can't split it up any further. The whole team's name is the team name, and that is the end of that.

The second kind of attribute is a **composite attribute**. These attributes “consist of a hierarchy of attributes”. (Watt & Eng, 2014) In the team example, the location of the team is just such a composite attribute. The location consists of the city and the country where the team is located. It would be best to split these two things up into two separate, single-valued attributes.

Thirdly, there are **multivalued attributes**. These are “attributes that have a set of values for each entity”. (Watt & Eng, 2014) In our example, we have the team colours which is a multivalued attribute. One team might have blue and orange as colours, another red and white, and a third maybe just black. So, lots of the team entity instances will have more than one colour.

If our design is only at the conceptual level of abstraction, having a multivalued attribute in an entity is perfectly fine. At the logical level, though, we will have to change it so that it can in fact be implemented.

The last type of attribute is a **derived attribute** – an attribute that can be calculated from the values of other attributes. (Watt & Eng, 2014) In the business rule that we are working with, we don’t have any derived attributes. But if we were to store the year when the team first entered, we could calculate the age of the team in years, based on the current year. That would be a derived attribute.

Derived attributes can either be stored in the database, or they can be calculated when they are needed. A trade off needs to be made between the two. Storing the derived attributes would mean that queries could be run much faster, at the expense of using more storage. But a major downside is that you need to make sure that the data stays up to date. (Blaha, 2016)

Now that we have the theory in place, we can add the attributes to the **Team** entity. For now, we will leave the **Colours** attribute as it is, just adding it to the entity.

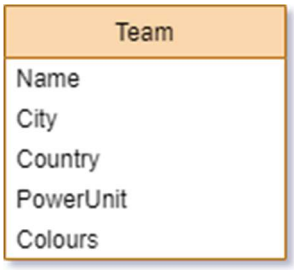


Figure 17. Business Rule 1 – Team Entity with Attributes

When it comes to naming attributes, do make the names descriptive. MySQL can have column names up to 64 characters long. So, there is no reason to use unnecessarily abbreviated names.

2.5 Relationships

Business Rule 2

A racetrack hosts multiple races over time. A racetrack can be stored even before the first race is scheduled to take place there. Each race takes place at a specific racetrack, on a specific date.

Now we have a new business rule, so let's start a new small ERD. When creating an actual diagram, you might choose to just add on to the first diagram. But for the purposes of this discussion, it is useful to see the rules in isolation. So, start a new diagram.

Step 3a is again identifying the entities that are involved here by looking for the nouns. Here we see a **racetrack** and a **race**.

Step 3b is identifying the relationship. Reading the rule again, we see that a racetrack hosts multiple races over time. The verb in that sentence is **hosts**. The verb is used as the name of the relationship unless it is something very generic like "has." Then a more suitable verb should ideally be found.

Step 3c is drawing a basic ERD. The most basic ERD would have, as we have come to expect entities and a relationship. So, let's draw the basic diagram.

Figure 18. Business Rule 2 – Basic ERD

Here we see that the two entities, Racetrack and Race, are represented. And they are connected by a line (called an association in UML) that indicates that they are related. Additionally, the name of the relationship appears together with an arrow that indicates the direction in which the verb should be read, in the form that it is written. This notation is proposed by (Coronel, et al., 2013).

Let's read back the relationship from the diagram to see whether it accurately represents what we wanted it to. Starting from the beginning of the arrow, a racetrack hosts a race. That sounds sensible. What if we wanted to read the relationship in the other direction, though? Then we would have to change the form of the verb to passive voice: A race is hosted by a racetrack. Yep, that works too.

Another way to represent the name of a relationship is to show both the verbs that describe a relationship on the same diagram. (Holowczak, n.d.) In our example, we do have a second verb – takes place at.

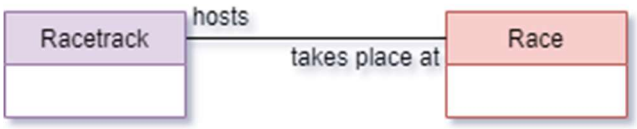


Figure 19. Different verbs on one relationship

Here we read the verb closest to the first entity. So, reading from left to right, a racetrack hosts a race. And reading from right to left, a race takes place at a racetrack. That works well, but it doesn't add value necessarily since "is hosted by" worked perfectly well. It then depends on the situation – sometimes people might insist that they only use a certain term in their organisation. And then it is worthwhile sticking to what they tell you.

Showing both verbs, however, is ideal if the verb wouldn't make sense in the opposite direction, for example, the case of a library that lends a book to a library user shown in Figure 20.

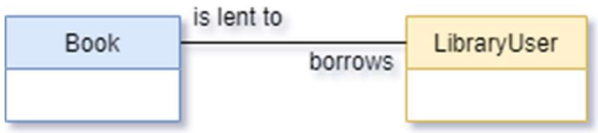


Figure 20. Different verbs on one relationship

Reading this from left to right, a book is lent to a library user. And the other way around, a library user borrows a book. This kind of situation doesn't arise very often, though, so for the most part, we will stick to the single verb with an arrow indicating which direction to read the verb as written.

There is more to business rule 2, though – it also indicates how many of each entity is involved in the relationship. A racetrack can host many races. And each race takes place at a specific racetrack. This is a one-to-many relationship.

Representing that requires using multiplicities!

2.6 Multiplicities

Multiplicity in a UML ERD is a combination of the **participation** and **cardinality** of the relationship. (Monge, n.d.)

Definition
"**Cardinality** refers to the maximum number of times an instance in one entity can relate to instances of another entity." (Lucid Software Inc., 2020)

Let us look at business rule 2 again:

Business Rule 2
A racetrack hosts multiple races over time. A racetrack can be stored even before the first race is scheduled to take place there. Each race takes place at a specific racetrack, on a specific date.

The maximum number of racetracks that are involved in the relationship is one. A race cannot take place at multiple racetracks, so the maximum is one. The maximum number of races that can be hosted by a racetrack is many. (A racetrack is expensive to build – it is desirable to use it more than once.) So, the **cardinality** of this relationship is said to be **one-to-many**.

Participation is about the minimum number of entities that are involved. This means that it can be **mandatory** or **optional** for one or both the entities. (University of Cape Town, 2017)

This rule is quite specific about the participation. It says that a racetrack can be in the database before the first race is scheduled. So, race is optional. But a race needs to take place at a specific racetrack. So, racetrack is mandatory.

Armed with all this information, we can now look at how to represent the multiplicities on the diagram. On each side of the relationship, we write the multiplicity as min/max. For example, if the minimum is 0 and the maximum is 1, then it would be 0..1. The minimum always comes first, no matter where it is placed on the diagram.

Values that are most frequently used for the minimum and maximum values are (Dybka, 2014c):

- 0 – zero
- 1 – one
- * – many

In UML notation, you could also get more specific, indicating that the maximum is a certain number rather than *. For example, if a student can have anywhere from zero to six modules in a semester, it can be written as 0..6. This kind of constraint is usually implemented in the software that makes use of the database, rather than the database itself.

So, now we can do Step 3d in Figure 14: add cardinality and participation.

Figure 21. ERD with Multiplicities

Now let us read the relationship from the diagram and see whether everything makes sense. The trick is to read the multiplicity closest to the second entity. And the way in which it is phrased is also important to avoid confusion. The phrasing proposed in (Monge, n.d.) is how we will read it here.

From left to right: Each racetrack hosts zero or more races. That sounds right. Initially, the racetrack exists in the database before the first race is held. And then many races are hosted there over time.

From right to left: Each race is hosted at one and only one racetrack. Yep, sounds good too.

Now, taking a step back, what type of relationship is represented here? We look only at the maximum values – 1 and *. So, this is a one-to-many relationship as we have said all along. Everything checks out.

Note that there is no rule about which way around entities can be drawn. It would be perfectly valid to have race at the top and racetrack below it, for example, if we keep the multiplicities and relationship names with an arrow the right way around. This is shown in Figure 22.

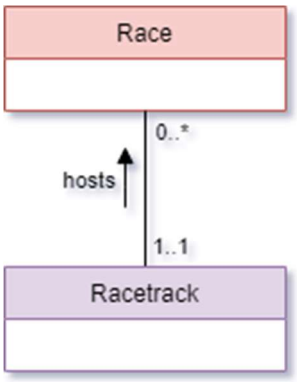


Figure 22. Vertical Diagram

2.7 Keys and Other Attributes

So far, the diagram that we have drawn is at the conceptual level of abstraction. We have represented just the entities and their relationships, and we haven't done anything yet that is specific to a type of database. Now that is about to change. The next step (Step 3e) is to identify primary and foreign keys, and that is in fact at the logical level of abstraction. This means that an ERD is not necessarily only a conceptual-level artifact – it depends on what information you put on the diagram.

For an excellent example of ERDs at the different levels of abstraction, read (Sezairi, 2019).

2.7.1 Primary Key (PK)

Before we jump into the specific example, let's look at the theory first. The first concept that we are talking about is the **primary key** (**PK**) that we need to choose for an entity.

Definition
"A key is chosen by the database designer to be used as an identifying mechanism for the whole entity set. This is referred to as the **primary key**." (Watt & Eng, 2014) "It must uniquely identify tuples [rows] in a table and not be null." (Watt & Eng, 2014)

Let us think for a moment about how people are identified. It is possible for two people to have the same name and surname. So, using that to identify a person is not unique enough if you are working with the whole country's population. That is why a unique ID number is allocated when a baby is born – that will stay the same even if the person's name is changed later. And two people will not officially be allocated the same number.

A similar unique identifier needs to be identified for each entity in the database. How do we uniquely identify a product? We use its PK. How do we uniquely identify a customer? We use that entity's PK.

There are several ways to categorise keys. The first way is whether it is a simple key or a composite key. A **simple key** consists of only one attribute (column). (Watt & Eng, 2014) The ID number of a person is an example of a simple key.

A **composite key** consists of two or more attributes (columns) that together uniquely identify the entity. (Watt & Eng, 2014) Say for example we wanted to store the details of all the students across all the tertiary institutions in the country. We could not use just the student number as the unique identifier, because a student at Varsity College might have the same student number as a student studying at some other university. If we use the student number and the institution together though, we can uniquely identify a student. That is a composite key.

The second way to categorise keys is whether it is a natural key or a surrogate key. A **natural key** is a unique identifier that naturally is found in the dataset. The ID number of a person, and the ISBN number of a book, are examples of this kind of key. (Larsen, 2011)

A **surrogate key** is a value that is generated and then used as the PK. It is usually just a number, and it doesn't have any meaning to the user of the system. (Larsen, 2011) So, a book in the database might get assigned sequentially the next number that may be 187623 for example.

The advantage of using a natural key is that it is easier and faster to search for, say, a book by its ISBN number. This would be what the user of the system would be familiar with and would want to search by. The disadvantage of using a natural key is that it might change as business requirements change. (Larsen, 2011) Think of licence plates for cars in South Africa. The format for licence plates in Gauteng used to be ZZZ 999 GP, until we ran out of numbers.

Now it is ZZ 99 ZZ GP. So, if a database and the software that uses it expected the first format, it would have to be changed to also accept the second format.

The advantages of using a surrogate key are that the key structure won't ever need to change, and they require very little storage space if they are integers (only four bytes each). (Larsen, 2011) And queries with joins are simpler if you use a single surrogate key rather than a composite natural key. (Coronel, et al., 2013) The disadvantage is that searching for the data requires a little more work. (Larsen, 2011)

Read (Larsen, 2011) for more details about the pros and cons of using natural and surrogate keys.

The recommendation for this module is to **always use integer surrogate keys**. (The notable exception is when doing normalisation – we will see why in Section 3.)

Why do we want to have a PK? Intuitively, we want to be able to uniquely identify an entity. But the reason behind that is that we want to preserve entity integrity.

Definition

"**Entity Integrity** ensures that there are no duplicate records within the table and that the field that identifies each record within the table is unique and never null.

"The existence of the Primary Key is the core of the entity integrity. If you define a primary key for each entity, they follow the entity integrity rule." (databasedev.co.uk, 2015)

If we can't uniquely identify, say, a person in the database, we cannot guarantee that we don't have duplicate entries for the same person. And we can't guarantee that we can find the person again in the database either. Which really makes the data in the database quite useless.

Now we have all the theory behind us, and we can identify the PKs that we need. Each entity must have a PK, and we said that the recommendation is to always use a surrogate PK. So, `Racetrack` gets a PK called `RacetrackID`, and `Race` gets a PK called `RaceID`.

Including the name of the entity here will make the queries later much simpler to write. So that is highly recommended.

Let us add the PKs onto the ERD.

Figure 23. ERD with PKs included

2.7.2 Foreign Key (FK)

Now that we can identify each entity, the next question is: how do we represent the relationship in terms of keys? By making use of a foreign key (FK).

Definition

"A **foreign key** (**FK**) is an attribute in a table that references the primary key in another table OR it can be null. Both foreign and primary keys must be of the same data type." (Watt & Eng, 2014)

How would we be able to record which racetrack a race is held at? By storing the ID of the `Racetrack` in the `Race`.
What if we get the racetrack's ID wrong, and a racetrack with the relevant ID doesn't exist? Well, then our data again lacks integrity. This time specifically referential integrity.

Definition

"**Referential integrity** requires that a foreign key must have a matching primary key or it must be null. This constraint is specified between two tables (parent and child); it maintains the correspondence between rows in these tables. It means the reference from a row in one table to another table must be valid." (Watt & Eng, 2014)

We will see later in the module how to set up the constraints in our database to make sure that referential integrity is enforced. For now, it is enough to include the FK in the diagram.

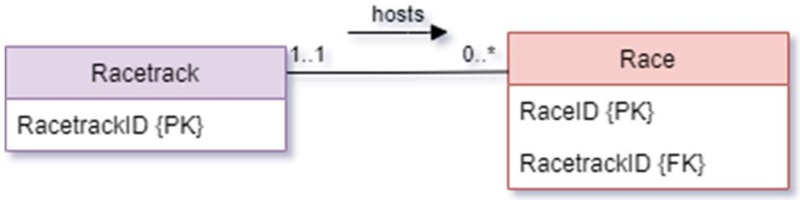


Figure 24. ERD with the FK included

Here is a very useful pattern to recognise: **In a one-to-many relationship, the FK always ends up on the many side.** Think about it for a moment. A race has a single racetrack where it takes place. So, if we put the FK in race, we are storing a single value. But if we tried to store the list of IDs of all the races hosted by a racetrack in the racetrack table, how would we store those multiple values? So, the FK should be on the many side.

2.7.3 Other Attributes

The last step in the process, Step 3g, asks us to identify and add other attributes. The rule specifically tells us here that a race takes place on a specific date. So, we can add date to the diagram.

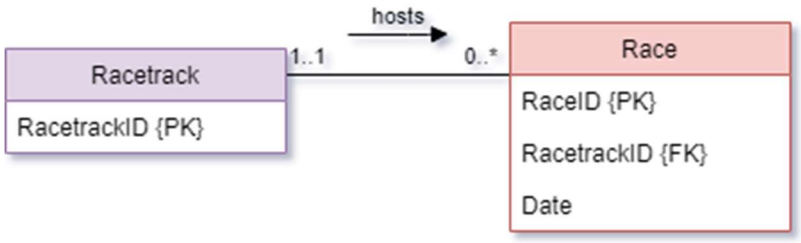


Figure 25. ERD with other attributes

We don't yet know what the other rules are going to be, so we don't really have more information about either `Race` or `Racetrack` to add right now. This lack of information does often occur in real business cases too, where you would have to ask analysts and users for more information. For now, what we have will do for business rule 2.

2.8 More About Relationships

We have now looked in detail at how to represent a one-to-many relationship in our ERDs. But there is more to know about relationships.

2.8.1 Multivalued Attributes

You might recall that we never did anything about the multivalued attribute in the first business rule. Now we have the knowledge to solve that problem. Where were we with that business rule?

Business Rule 1
When a team enters, the following information needs to be provided: team name, location of where the team is based, the power unit used by the team and the team colours.

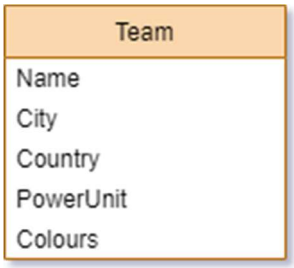


Figure 26. Business Rule 1 – Team Entity with Attributes

We can now make a second entity called `Colour`, with a one-to-many relationship with team. After all, one team can have many colours. Here we assume that the descriptions that we store are just going to be strings that can be anything, including something like "Ferrari red". Under this assumption, multiple teams won't have the same colour.

Tip: It is always useful to state assumptions when drawing ERDs.

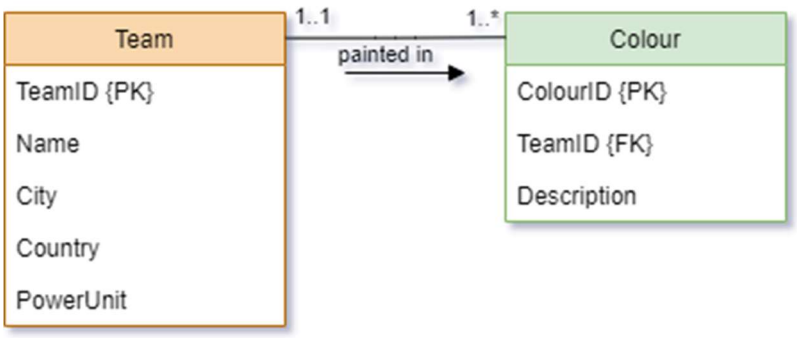


Figure 27. ERD showing Team and Colour

2.8.2 Other Cardinalities

Business Rule 3

Each team has several staff members (that are not drivers). A staff member is only allowed to ever work for one team. The name and surname of each staff member needs to be stored in the database, as well as the date that they joined and (optionally) the date that they left the team. One of the staff members is the team chief.

Here we have two entities: `Team` and `StaffMember`. There are two interesting things about this rule. The first is that the rule in fact specifies two different relationships between the same two entities. Let's first draw the one-to-many relationship between `Team` and `StaffMember`.

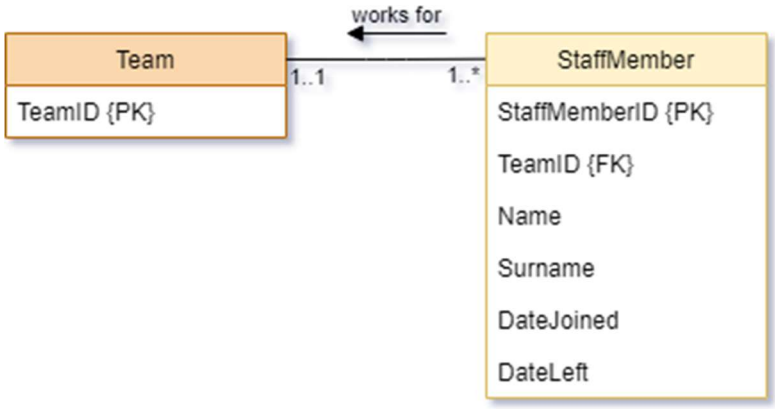


Figure 28. Each StaffMember works for a Team

This relationship follows the standard pattern for a one-to-many relationship: the FK appears on the many side. We assume here that a team must at least have one staff member. It is a good assumption to make, since we know that there is a position of team chief that needs to be filled.

The second relationship is the fact that one of the staff members is the team chief. Let's think a little about the multiplicities here. A team will have at most one team chief, and it doesn't say that it is optional, so we can assume that the minimum is also one.

A staff member can be the team chief for at most one team (because they can't work for multiple teams to start with). But the minimum is interesting. Not all staff members are team chiefs, so that is optional, and the minimum is thus 0. With one-to-one relationships, there is often this implied optional side, so think carefully about them.

This is then the second interesting thing about this rule: it specifies a one-to-one relationship. These relationships are not encountered as often as one-to-many relationships.

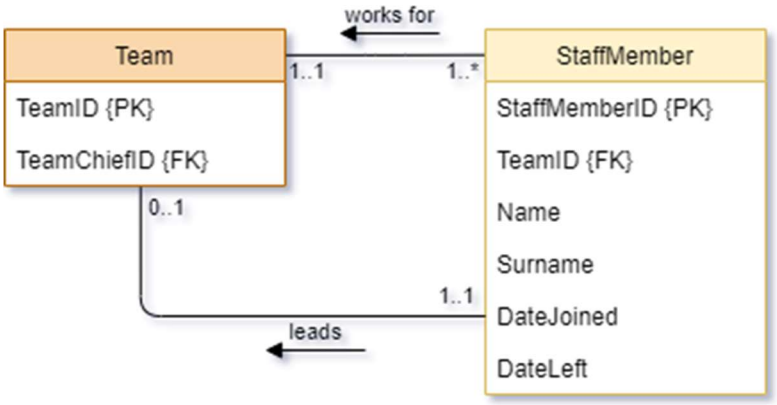


Figure 29. Second Relationship Added

If there is a relationship on a diagram, there must be a corresponding foreign key. Here we added an FK called `TeamChiefID` to the `Team` entity. It is worthwhile making the name of the FK descriptive here. If it was just `StaffMemberID`, you might wonder why it is there. But `TeamChiefID` leaves no doubt what it represents.

The FK could work on both sides here since it would be a single value regardless. But there is still a more optimal solution. If we put the `ChiefTeamID` FK in the `StaffMember` table, there would be lots of staff members with null values in that attribute. That would use space for no reason. But if we put the `TeamChiefID` in the Team, then all the teams would have a value.

So, the guideline is: **If there is a one-to-one relationship with one side optional, put the FK on the optional side.**

If both sides were mandatory, where the FK appeared would have depended on what made more sense from a business perspective.

If you are a motorsport fan, you are probably asking by now, "but what about the DRIVERS?!" Coming up next.

Business Rule 4

Each team can have two drivers at a time. Drivers may decide to change teams, and the history of that needs to be stored in the database. The date joined that a driver joined a team and (optionally) the date that they left needs to be stored. The name and surname of each driver needs to be recorded.

Here, things get very interesting indeed. We have two entities: `Team` and `Driver`. And the relationship we can call 'drives for'. That is all still straight-forward. But what about the relationship here? Well, a team can, over time, has many different drivers. And a driver can decide to change teams. (It has happened, even mid-season!) So, over time, a driver can also have many teams. This is a many-to-many relationship. Let's draw that and see what it looks like.

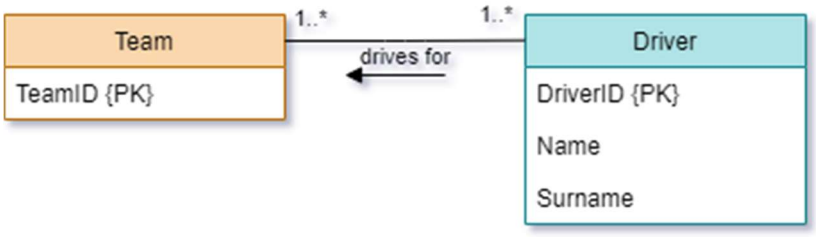


Figure 30. Many-to-many Relationship

Having a many-to-many relationship would be fine if we were only modelling at the conceptual level of abstraction. There is no UML notation for specifying properties on relationships, though, so there isn't a way to indicate that the dates need to be stored.

We do, however, want to design down to the logical level. And we have a problem here in terms of representing the FKs. Just like putting the FK in a one-to-many relationship on one side would result in this awkward list of FKs that we need to do something with, we now have that problem on both sides of the relationship. Step 3f gives us a hint – we need to do something to eliminate the many-to-many relationship.

The way to eliminate the many-to-many relationship is to add a bridging entity. It is an entity that is going to contain the matching IDs for each combination of driver and team. Once we create the table, we will have a row, for example, for driver 8 that drives for team 3. And then maybe next year, driver 8 drives for team 2 instead.

The bridging entity will also be used to store the attributes of the relationship – here the date joined and the date left. When there are other attributes like we have here, there is a strong case to be made that the bridging entity should also have its **own surrogate PK**.

We need to find a good name for this bridging entity. Sometimes you can make use of the verb to find a good name. But here anything that sounds like 'drives' would be too close to `Driver` and will just cause confusion.

Reading about it online, it emerges that drivers have a contract with the team. So, let's call it `Contract`. If you work in an organisation, find the term that the organisation uses!

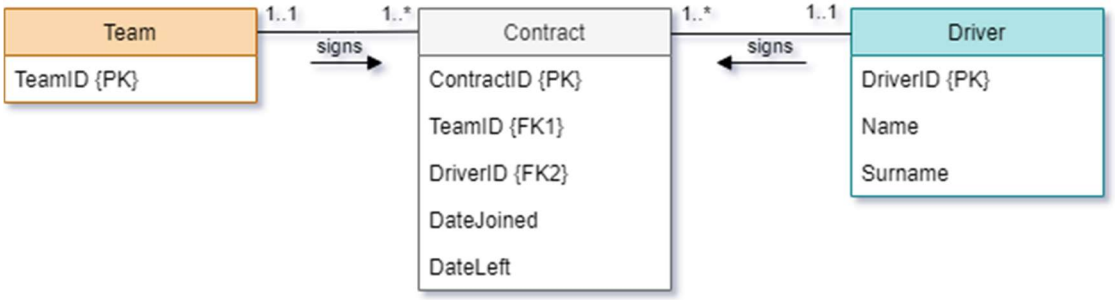


Figure 31. Bridging Entity Contract

When a bridging entity is introduced, the many-to-many relationship is split into two one-to-many relationships. **And the pattern is that the many sides of those relationships are next to the bridging entity**. Let's read the new relationships to see whether that makes sense. Each team signs many contracts. Each contract is signed by one team. Yup, that sounds sensible. Now let's look at the driver's side. Each driver signs many contracts. Each contract is signed by one driver. That also makes sense.

Looking at the whole thing taken together though, it still represents the fact that many teams sign contracts with many drivers.

Notice that the two FKs are numbered here, to represent the fact that they link to the PKs of two different tables.

2.8.3 Degree of Relationship

There are two different ways in which to classify relationships: by cardinality (as we have seen so far) and by degree of the relationship. The degree of the relationship is the number of entities participating in the relationship. (Pearson Education, 2009) So far, we have been working with **binary relationships** – relationships between two entities. But there are also other possibilities:

- A **unary relationship**, also called a **recursive** relationship, is a relationship where an entity instance is related to other instances of the same entity. For example, one employee that manages other employees. (Watt & Eng, 2014)
- A **ternary relationship** is one where three different entities are involved. For example, a supplier supplies a part for a specific project. (Watt & Eng, 2014) For the relationship to make sense, the supplier, the part, and the project needs to all be specified. So, a ternary relationship is more than just a collection of one-to-many relationships. It is really three entities that are used together to represent one transaction.
- A **quaternary relationship** is a relationship where there are four entities involved. (Pearson Education, 2009) These relationships are quite rare.

Let us look at an example of a unary relationship.

Business Rule 5

One staff member is designated as the crew chief. The crew chief manages other staff members that work in the pit crew.

This business rule makes it quite clear that we have one staff member who manages other staff members. So, only one kind of entity is involved here: `StaffMember`. That makes this a **unary relationship**.

The one staff member manages multiple other staff members, so it is also a one-to-many relationship. Not all staff members are managers, so it is optional on one side.

Also, not all staff members will have a manager in this database, since there will be top-level people that won't have their managers recorded here. The team chief might report to somebody in the organisation that sponsors the team. But that reporting line would not be recorded in the context of the larger competition. So, for this database we do need to make having a manager optional.

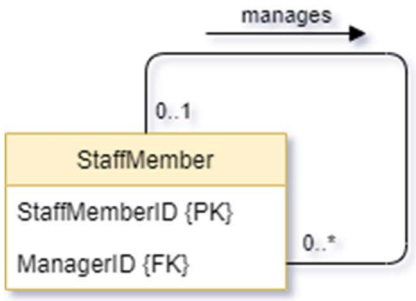


Figure 32. Unary Relationship

Here, the arrow next to the verb is essential, to indicate not only the direction in which the verb is read but also how that relates to the multiplicities. Each staff member manages zero or more other staff members. And each staff member is managed by zero or one other staff member. So, we see that which way around we place the multiplicities does make a difference.

Although it is not as obvious here, the FK does still go on the many side. So, the ID of the manager is stored for each team member.

What does a ternary relationship look like?

Business Rule 6

A driver participates in a specific race, racing for a specific team. The position that the driver places in that race needs to be recorded in the database. A driver can compete in many races over time.

Now things are starting to come together! The points awarded to the driver in a race not only counts towards their own ranking but also towards the team (that is called constructor standings). So, driver, team, and race all need to be connected in one single relationship for this data to be represented.

Rephrasing the rule with the knowledge that we have from previous rules: a driver can participate in many races for many different teams over time. So, this is a many-to-many relationship with three entities, and hence a ternary relationship.

Why can we not just look at which team the driver is on, and use that to determine the team? (Then we would have two binary relationships, which feels simpler.) Because drivers can change teams. So, it gets a bit more complicated – we would have to look at the dates too. Which team did the driver belong to on the date that the race took place? To avoid that complexity, let us just use the ternary relationship that contains all the relevant data.

Just like with a many-to-many binary relationship, we also need to introduce a bridging entity to contain the keys of the other entities that are involved. There is one attribute that we need to store in that bridging entity too: position.

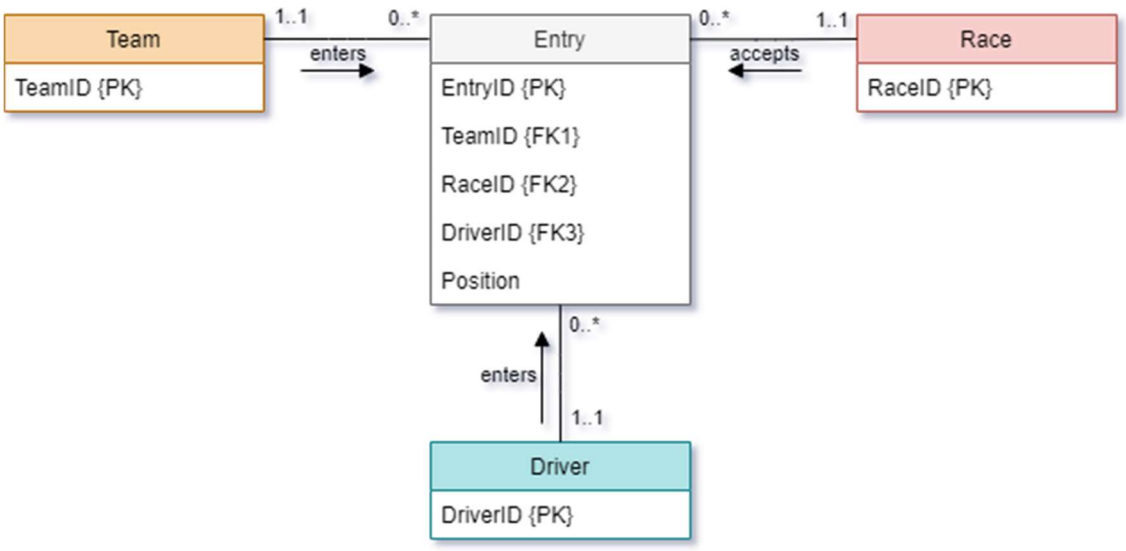


Figure 33. Entries for a Race

Here, the multiplicities next to `Entry` indicate that an entry is optional. This is because a race would have to be created first, and then it could accept entries. Similarly, for the other entities, we assume that they could exist in the database without having any entries yet.

2.8.4 Combining the Diagrams

There are probably many more business rules in F1 racing that we have not documented yet. But so far, we have covered everything that we needed to discuss. So, let's combine the small diagrams into one large diagram.

First, let's get all the diagrams together on one page so we can see what we are working with.

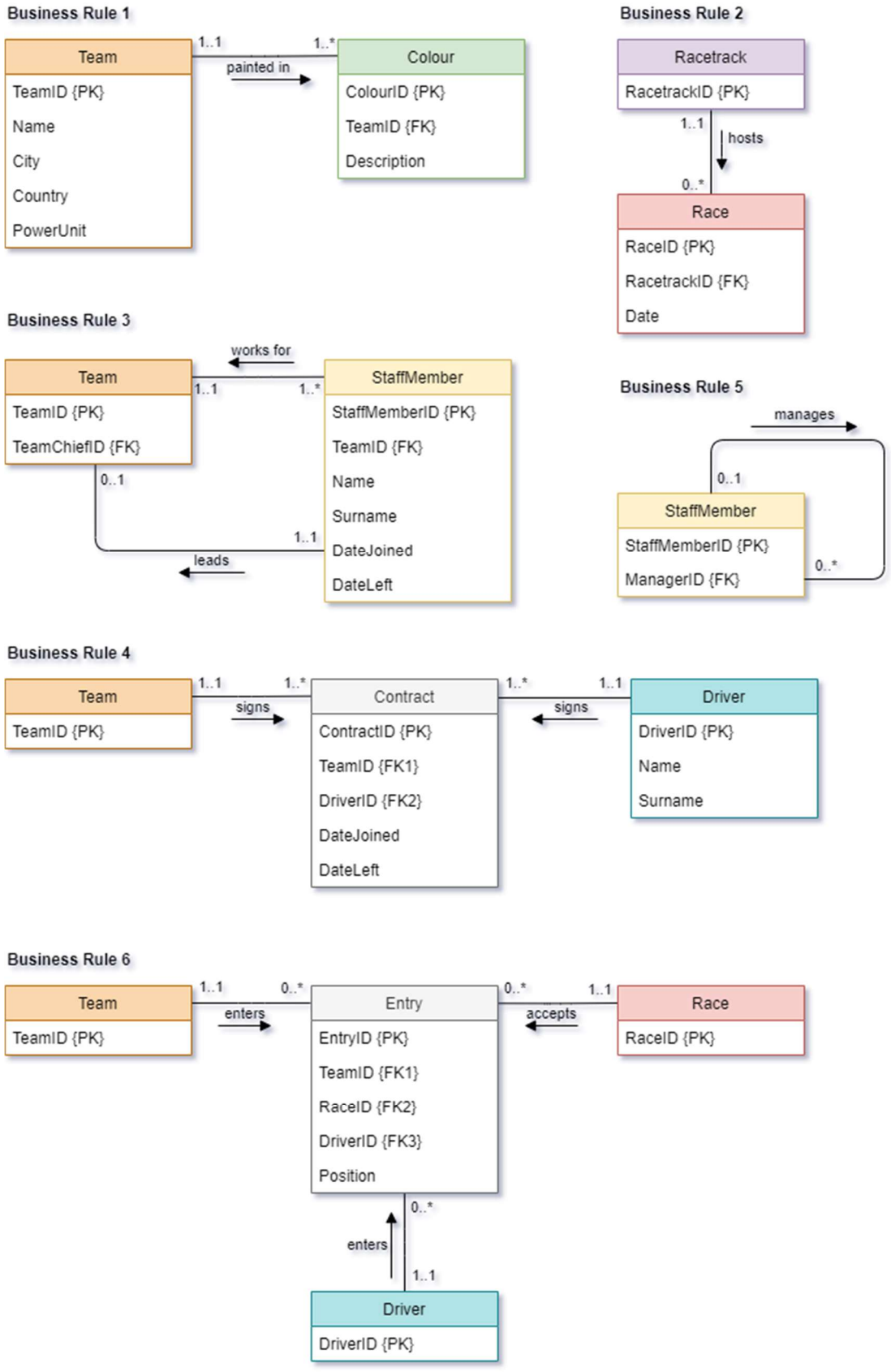


Figure 34. All the ERDs

Seeing it all together like this is a little daunting. Where would we even start with combining these? Figure 14 has some steps to follow that will be helpful:

4. Combine small diagrams into a single ERD.
    4a. Find the most frequently used entities.
    4b. Position these entities centrally.
    4c. Start adding other entities and relationships, making sure you add any foreign keys (numbered) as needed.
    4d. Add multiplicities and relationship names at the end to avoid having to move these around.

Figure 35. Combining Small Diagrams

Counting the number of times each entity appears, we see that Team is used four times, so that is currently the most central entity. Then `StaffMember`, `Driver`, and `Race` each appears twice. So, by putting those entities in central places, we should be able to draw a diagram where the relationship lines don't need to cross.

There is one additional improvement that can be made, and that is adding a name attribute for the racetrack. The combined ERD is shown in Figure 36 on p.60.

The drawback of having lots of small ERDs to combine is that the process can be a little error-prone. That makes it even more important to check everything once the diagram is complete.

2.9 Checklist

The last phase in the process shown in Figure 14 is reviewing and revising the ERD. When working on a real project, the various stakeholders would be involved in this review process. When the analysts and users who specified the business rules in the first place see the diagram, they can quickly spot not just errors but also omissions – things that never were in the business rules to start with.

For our purposes, we don't have access to business users to give us feedback. But there are some final checks that you can do to make sure that the ERD that you have created correctly reflects the business rules as stated. The checklist is shown in Figure 37.

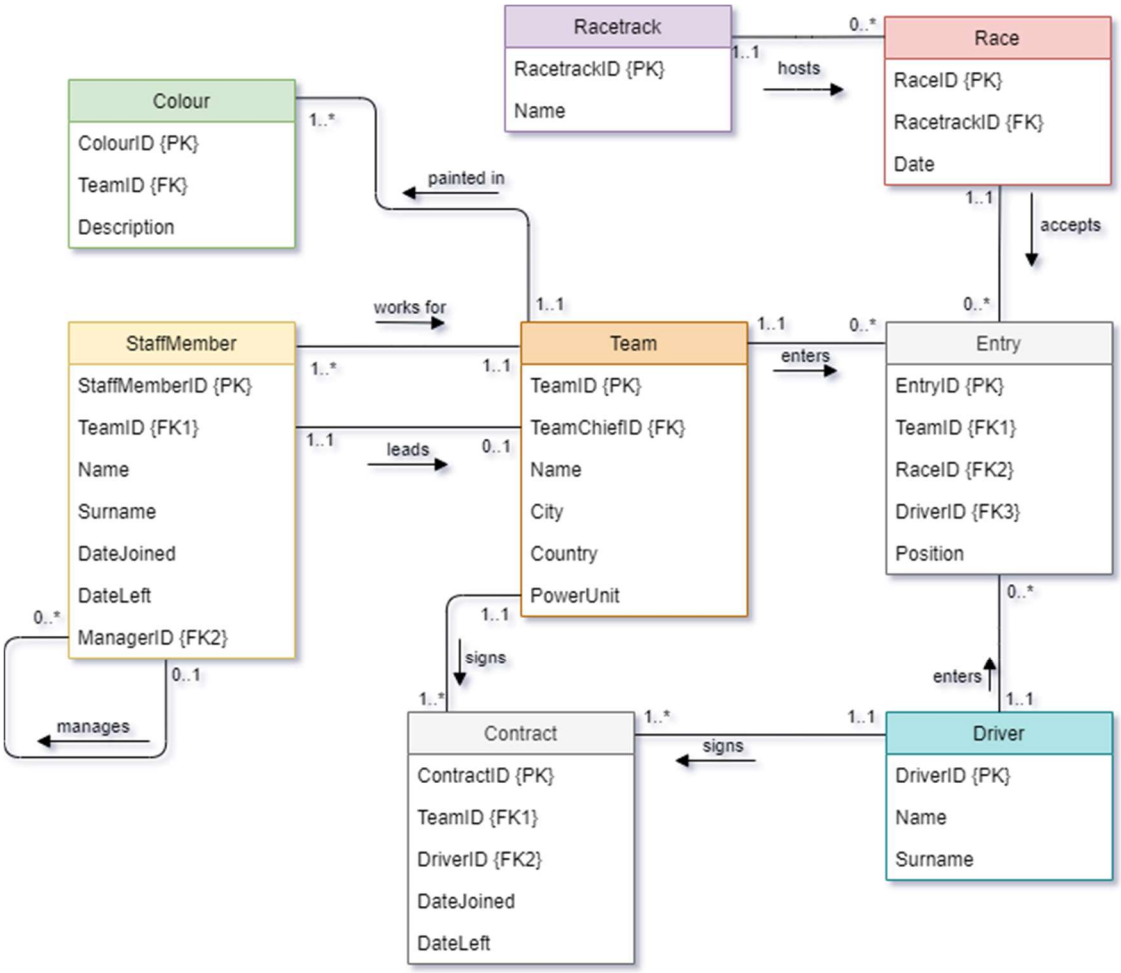


Figure 36. Combined Diagram

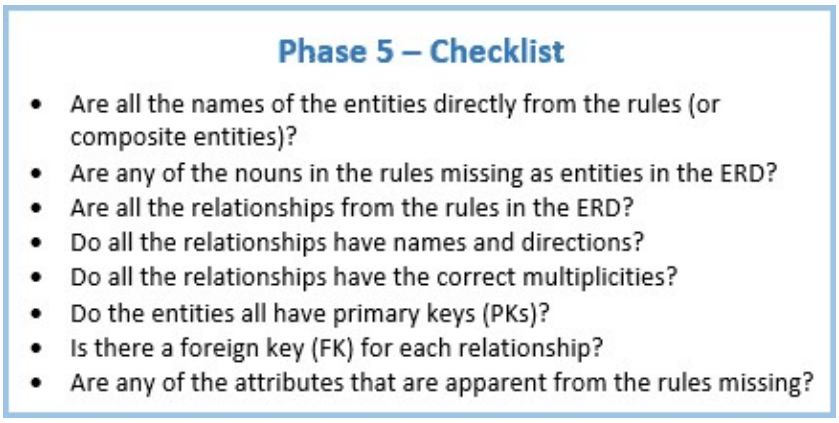


Figure 37. ERD checklist (Pellissier, 2019)

When it comes to checking the relationships, especially the multiplicities, the best way is to read back the relationships as depicted on the diagram.

And then compare those to the given business rules. That will catch most of the obvious cases where multiplicities are the wrong way around.

3 Normalisation

The information about normalisation is adapted from (Watt & Eng, 2014, pp. 63-83), unless otherwise indicated.

3.1 What is Normalisation?

We have now looked at how to create an ERD based on business rules. And that is an essential part of the database design process since it provides a big-picture view of the data requirements.

Now we get to normalisation, which is another part of the design process that should be used concurrently with the ERDs. This process goes into a more detailed view of specific entities. Where ERDs contain the macro view, normalisation gets into the micro view. By using normalisation, we can improve the design of the attributes of our entities.

Definition
Normalisation “is the process of determining how much redundancy exists in a table.” (Watt & Eng, 2014)

The fact that a table is mentioned in this definition, points out that normalisation works based on actual data rather than business rules. That again shows that we get into the details here.

The goals of normalisation are to determine whether there is redundant data in our design, and then find a way to remove those redundancies.

3.1.1 Functional Dependency

A central concept in normalisation is a functional dependency. So, let us explore that first.

Definition
“A **functional dependency** (**FD**) is a relationship between two attributes, typically between the PK and other non-key attributes within a table.” (Watt & Eng, 2014)

The term **functional dependency** is a good one, since it avoids making use of the word relationship which can cause confusion. So, we will stick to **dependency** throughout this manual.

Let us look at a practical example – the Driver entity from our Formula 1 ERD:

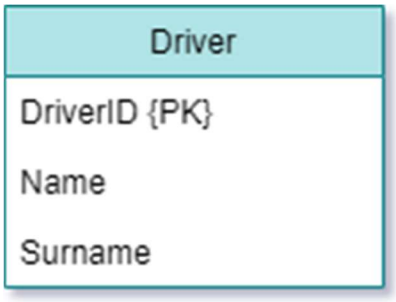


Figure 38. Driver Entity

In the ERD discussion, we said that the PK is used to uniquely identify a specific driver. That is the very purpose of having the PK. We can then say that there is a functional dependency where the `Name` and `Surname` attributes are dependent on the `DriverID`. We can write that as follows:

`DriverID → Name, Surname`

If we look up a driver by `DriverID`, that PK will determine which `Name` and `Surname` we will get back from the database. The `DriverID` is called the *determinant*, and the `Name` and `Surname` are called *dependent* attributes.

Let us look at some sample data to explain that further. These are all drivers from the F1 Hall of Fame. (Formula One World Championship Limited, 2020)

Table: Driver

DriverID	Name	Surname
1	Michael	Schumacher
2	Lewis	Hamilton
3	Nico	Rosberg

Figure 39. Sample data for Drivers

If we were to ask who the driver with `DriverID` 1 is, we would see that we get the `Name` Michael and the `Surname` Schumacher. So, the `Name` and the `Surname` attributes depend on the `DriverID`.

Notice in the above table that the PK column's name is underlined, and the name of the table is shown. These are both good conventions to stick to when working with tabular data.

3.1.2 Introduction to Dependency Diagrams

Representing the column dependencies in the text format with the arrow is all fine and well. But there is a visual way to represent the same thing, called a dependency diagram.

Figure 40. Driver dependency diagram

In this diagram, the arrows indicate the dependencies between the attributes. The arrow starts at the determinant and ends at the dependent attributes. So, the arrow points in the same direction as in the text representation.

Using dependency diagrams is the easiest way to master normalisation (Czenky, 2010), and it is also recommended in (Coronel, et al., 2013).

3.2 Benefits of Normalisation

The benefits of normalisation include:

- **Better organisation** of the data in the database. (Plew & Stephens, 2003)
- **Reduced data redundancy**, which means that the data in the database will be more consistent. (Plew & Stephens, 2003)
- Because there is less repeated data, the database takes up **less storage space**. (Import.io, 2019)
- **Operations** like searching, sorting, and creating indexes are **faster** because the resulting tables contain fewer columns. (Sybase Inc., 2003)

3.3 First Normal Form

There are six normal forms, but we will only discuss the first three. The **Third Normal Form** (**3NF**) is considered the best trade-off for transactional databases between data consistency and performance. (Fuller, 2006) So, we won't go further than that.

First Normal Form (**1NF**) is defined as data where "only single values are permitted at the intersection of each row and column; hence, there are **no repeating groups**." (Watt & Eng, 2014). Another way of putting it is that a "relation is in first normal form if **every attribute** in that relation is **singled valued attribute**." (Upadhyay, n.d.)

Additionally, a **composite primary key** must be identified, that all the other columns will depend on. (Coronel, et al., 2013)

Side note: There is some disagreement between different authors about how far to go for 1NF. We will stick to the definition as used by (Coronel, et al., 2013), (Upadhyay, n.d.) and (www.softwaretestinghelp.com, 2020). So, if you want to read more about normalisation, especially 1NF, make use of those sources. Many other sources online will only confuse you with slightly different definitions.

Figure 41 shows the whole process that we are going to be following to do the normalisation by using dependency diagrams.

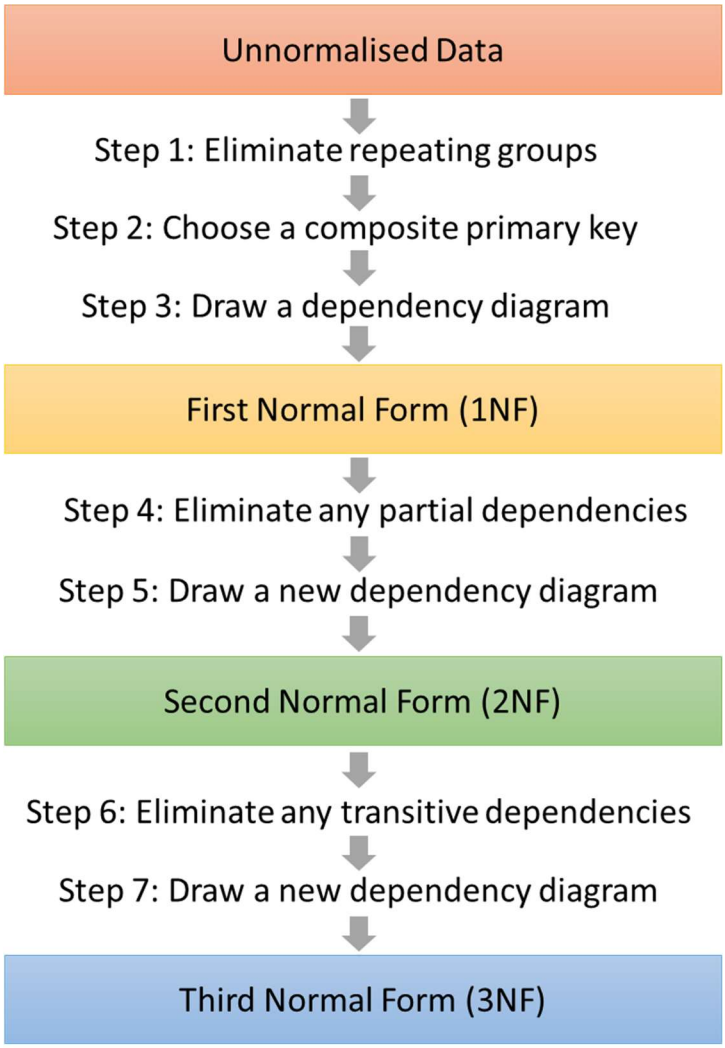


Figure 41. Normalisation Process, based on the process as described by (Coronel, et al., 2013)

Unnormalised data is in the form of actual, sample data. There is no real way to represent that in either a dependency diagram or an ERD. From **1NF** onwards, we will be using **dependency diagrams as the main way to represent the design**, but it could also be shown in table format or even as ERDs. So, we will also show here those alternative ways for completeness.

Let us look at an example of unnormalised data so we can see what repeating groups and multi-valued attributes look like. This is a small extract from the racing results for the 2020 season.

Driver	Driver Country	Driver DoB	Race	Racetrack	Racetrack City	Position
Sebastian Vettel	Germany	03/07/1987	Hungary 2020	Hungaroring	Budapest	6
			Austria 2020	Red Bull Ring	Spielberg	10
Charles Leclerc	Monaco	16/10/1997	Hungary 2020	Hungaroring	Budapest	11
			Austria 2020	Red Bull Ring	Spielberg	2
Kimi Räikkönen	Finland	17/10/1979	Hungary 2020	Hungaroring	Budapest	15
			Austria 2020	Red Bull Ring	Spielberg	NC
Antonio Giovinazzi	Italy	14/12/1993	Hungary 2020	Hungaroring	Budapest	17
			Austria 2020	Red Bull Ring	Spielberg	9

Figure 42. Formula 1 data, derived from https://www.formula1.com/ [Accessed 8 December 2022].

3.3.1 Step 1: Eliminate Repeating Groups

Repeating groups are the places where there are multiple values in a single row. Another way of looking at it, is where there are merged cells in the original table. For every driver, there are multiple race results recorded here. So, those results are repeating groups.

As it stands, this data could not be stored in a database table. How would you represent the data without losing something if you could only store a single value in each row/column intersection?

To remove the repeating groups, we need to fill in the blanks in the table. This feels counterintuitive since the aim of normalisation is to remove redundancy. But stick to the process, and all will become clear.

Let us duplicate the driver details so that we have a whole row where each attribute has a single value. This is shown in Figure 43.

Driver	Driver Country	Driver DoB	Race	Racetrack	Racetrack City	Position
Sebastian Vettel	Germany	03/07/1987	Hungary 2020	Hungaroring	Budapest	6
Sebastian Vettel	Germany	03/07/1987	Austria 2020	Red Bull Ring	Spielberg	10
Charles Leclerc	Monaco	16/10/1997	Hungary 2020	Hungaroring	Budapest	11
Charles Leclerc	Monaco	16/10/1997	Austria 2020	Red Bull Ring	Spielberg	2
Kimi Räikkönen	Finland	17/10/1979	Hungary 2020	Hungaroring	Budapest	15
Kimi Räikkönen	Finland	17/10/1979	Austria 2020	Red Bull Ring	Spielberg	NC
Antonio Giovinazzi	Italy	14/12/1993	Hungary 2020	Hungaroring	Budapest	17
Antonio Giovinazzi	Italy	14/12/1993	Austria 2020	Red Bull Ring	Spielberg	9

Figure 43. Repeating Groups Removed

3.3.2 Step 2: Choose a Composite Primary Key

This is the most important, and often most difficult, step of the whole normalisation process. We need to find a combination of values that are already in the data, that will uniquely identify each row. Or, putting it another way, we need to determine which columns would form a composite primary key made up of natural key columns.

If we had the name of the driver, would we be able to uniquely find a specific row? No, there are duplicates now that we have eliminated the repeating groups. What about if we knew the race? Nope, there are duplicates of those too.

But what if we knew the name of the driver, and the name of the race? That will work! Each driver can only enter a specific race once, so the combination of those two items will uniquely identify the row. We have found our composite primary key.

This example is not particularly complicated. But we already know something about Formula 1 by now, and we have good sample data. In a real-world business problem, you might have to ask business users for clarification when identifying the composite PK.

Now we can indicate the PK columns on the table.

Table: DriverPosition
Primary key: Driver, Race

Driver	Driver Country	Driver DoB	Race	Racetrack	Racetrack City	Position
Sebastian Vettel	Germany	03/07/1987	Hungary 2020	Hungaroring	Budapest	6
Sebastian Vettel	Germany	03/07/1987	Austria 2020	Red Bull Ring	Spielberg	10
Charles Leclerc	Monaco	16/10/1997	Hungary 2020	Hungaroring	Budapest	11
Charles Leclerc	Monaco	16/10/1997	Austria 2020	Red Bull Ring	Spielberg	2
Kimi Räikkönen	Finland	17/10/1979	Hungary 2020	Hungaroring	Budapest	15
Kimi Räikkönen	Finland	17/10/1979	Austria 2020	Red Bull Ring	Spielberg	NC
Antonio Giovinazzi	Italy	14/12/1993	Hungary 2020	Hungaroring	Budapest	17
Antonio Giovinazzi	Italy	14/12/1993	Austria 2020	Red Bull Ring	Spielberg	9

Figure 44. Repeating Groups Removed

This is good enough in terms of data for 1NF. In this format, the data could be physically stored in one big database table. But remember, we said that we want to work with dependency diagrams, so let's see what that looks like in 1NF.

3.3.3 Step 3: Draw a Dependency Diagram

To create the **dependency diagram**, we start by drawing all the fields. Then we add the functional dependencies on the composite PK at the top, the partial and transitive dependencies at the bottom. All the other fields must be dependent on the composite PK; otherwise, we have missed something when choosing the key. So that goes at the top.

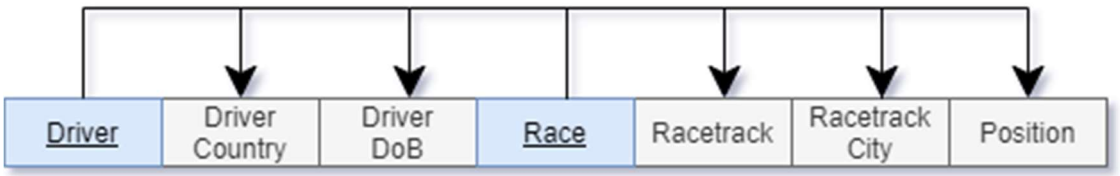


Figure 45. Dependency Diagram showing the Functional Dependencies

Note that the composite PK column names are underlined here. Remember to do that since it will help when identifying the partial and transitive dependencies later.

A **partial dependency** is where the value of an attribute is dependent on only part of the composite primary key. (Watt & Eng, 2014) In the sample data, do you need to know both the `Driver` and `Race` to determine which country a driver is from? No, the `Race` has nothing to do with that. So, the first partial dependency that we see here is:

`Driver → Driver Country`

The same is true for the date of birth (DoB) of the driver – it is also only a partial dependency.

Hint: An easy way to spot some of these partial dependencies is to look at the original unnormalised data.

If we knew only the `Race`, we would be able to find the `Racetrack` and `Racetrack City`. So, those are also partial dependencies.

What about `Position`? Could we know that by looking at either the `Driver` or `Race` in isolation? No, it is in fact fully dependent on the full composite PK.

Let us add the partial dependencies to the diagram – below the fields this time. Notice that position doesn't have a partial dependency on the diagram!

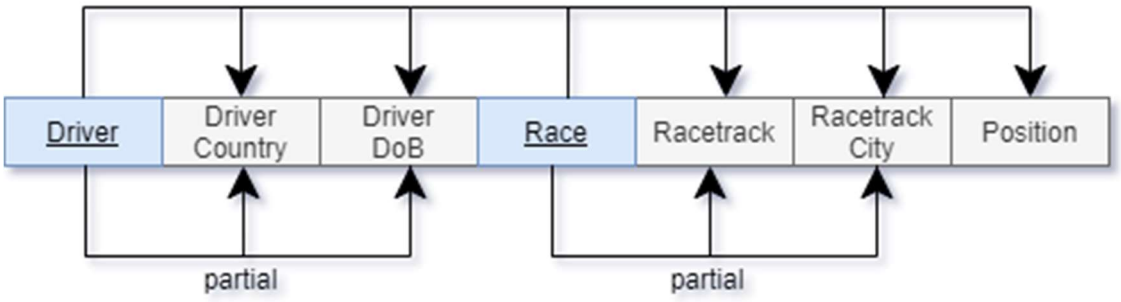


Figure 46. Dependency Diagram showing the Partial Dependencies

The last kind of dependency that we need to identify is a transitive dependency. A **transitive dependency** is a dependency where the determinant is not a primary key field at all. (Coronel, et al., 2013) The example in this data is `Racetrack City`. If we knew which `Racetrack` the race is held at, we could determine (at least in the sample data) which city it is located in. So, that is a transitive dependency. Let's add that to the diagram too, to complete it.

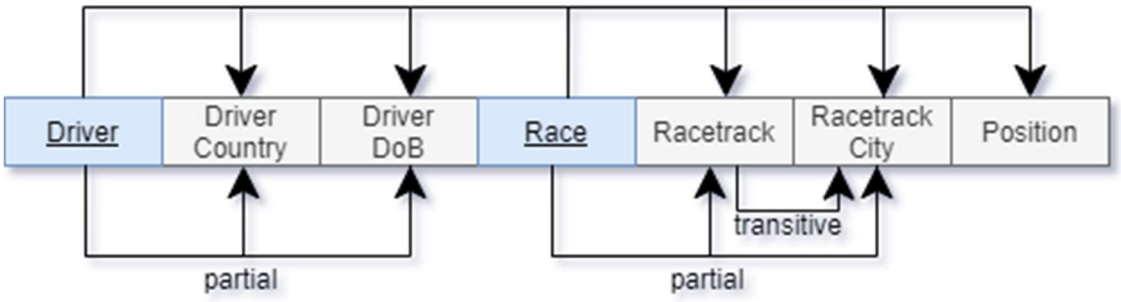


Figure 47. Final Dependency Diagram in 1NF

In this dependency diagram, we then see all the different dependencies – full, partial, and transitive. That represents the state that the data is in now in 1NF, and all of that is perfectly fine in terms of the definition of 1NF.

3.4 Second Normal Form

Definition

"For the **second normal form**, the relation must first be in 1NF. The relation is **automatically** in **2NF** if, and only if, the **PK** comprises a **single attribute**.

"If the relation has a **composite PK**, then each non-key attribute must be **fully dependent** on the entire PK and not on a subset of the PK (i.e., there must be **no partial dependency** or augmentation)." (Watt & Eng, 2014)

Now we continue to 2NF. The first requirement is that we must already be in 1NF. Done. The second requirement is that we need to eliminate the partial dependencies.

3.4.1 Step 4: Eliminate Any Partial Dependencies

This might sound daunting, but it is easy once you spot the pattern. For each column that forms part of the composite PK, we are going to create a new table.

This is where the dependency diagram really comes in handy. Look at Figure 48. Following the arrow from the `Driver` attribute, we know which attributes are dependent on it: `DriverCountry` and `DriverDoB`. So, the area outlined in a dotted line and coloured purple will all be part of a new table.

We can follow a similar pattern for `Race` too, that determines `Racetrack` and `RacetrackCity`.

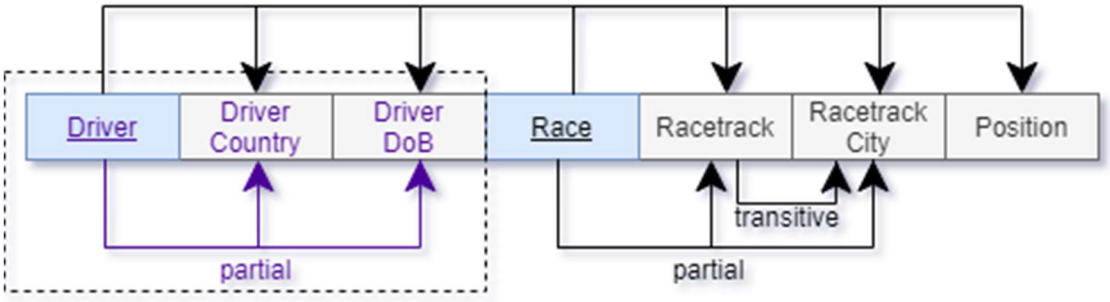


Figure 48. Diagram with Driver Highlighted

The important gotcha in this process is that we need to create a third table which contains the fully dependent fields – `Position` in this case.

We can represent these three tables in a text format like this:

```
Driver (Driver, DriverCountry, DriverDoB)
Race (Race, Racetrack, RacetrackCity)
DriverRace (Driver, Race, Position)
```

Notice that once again, the PK fields are underlined.

Let's revisit the definition to see whether this is in fact now in 2NF.

> "The relation is **automatically** in **2NF** if, and only if, the **PK** comprises a **single attribute**." (Watt & Eng, 2014)

`Driver` and `Race` both have a single attribute as the PK, so those are happily now in 2NF. What about `DriverRace`?

> "If the relation has a **composite PK**, then each non-key attribute must be **fully dependent** on the entire PK and not on a subset of the PK." (Watt & Eng, 2014)

Is `Position` fully dependent on both the `Driver` and `Race`? Yes, we have already determined that it is. So, `DriverRace` is also in 2NF.

The pattern that we have here might feel familiar by now – a table that contains the PK of two other tables … `DriverRace` is a linking table (or bridging entity). So, what is represented here is a many-to-many relationship with the bridging entity already introduced. It makes perfect sense if you think about it. A driver participates in many races, and each race is entered by many drivers.

3.4.2 Step 5: Draw a New Dependency Diagram

Now we can draw a new dependency diagram that shows the three tables we have created, with all the dependencies.

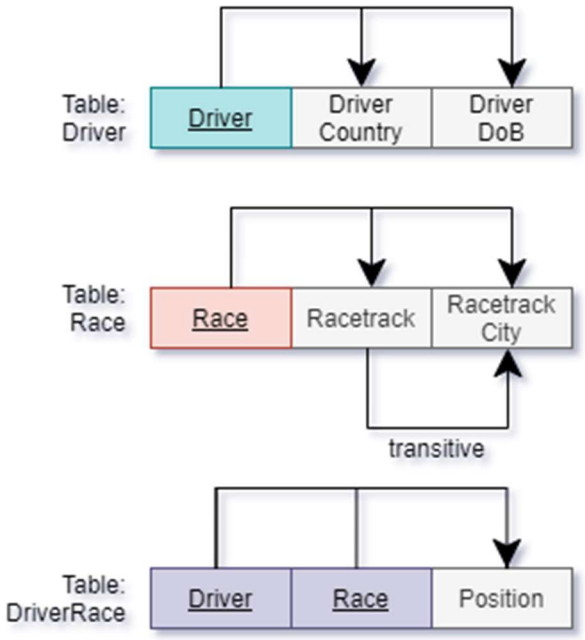


Figure 49. 2NF Dependency Diagram

`Driver` and `DriverRace` now have only full dependencies, so those appear above the attribute names now. In `Race`, the fields that are dependent on the PK are also shown above the attributes, but there is still one transitive dependency left:

`Racetrack → Racetrack City`

And that transitive dependency appears below the attributes.

3.4.3 Data in Table Format

What would the data look like now if we were to show it in table format?

Table: Driver
Primary key: Driver

Driver	Driver Country	Driver DoB
Sebastian Vettel	Germany	03/07/1987
Charles Leclerc	Monaco	16/10/1997
Kimi Räikkönen	Finland	17/10/1979
Antonio Giovinazzi	Italy	14/12/1993

Figure 50. Driver Table in 2NF

Table: Race
Primary key: Race

Race	Racetrack	Racetrack City
Hungary 2020	Hungaroring	Budapest
Austria 2020	Red Bull Ring	Spielberg

Figure 51. Race Table in 2NF

Table: DriverRace
Primary key: Driver, Race

Driver	Race	Position
Sebastian Vettel	Hungary 2020	6
Sebastian Vettel	Austria 2020	10
Charles Leclerc	Hungary 2020	11
Charles Leclerc	Austria 2020	2
Kimi Räikkönen	Hungary 2020	15
Kimi Räikkönen	Austria 2020	NC
Antonio Giovinazzi	Hungary 2020	17
Antonio Giovinazzi	Austria 2020	9

Figure 52. DriverRace Table in 2NF

We have already eliminated a lot of data redundancy. The only remaining data that might still be redundant is the `Racetrack City`. This is not apparent from the sample data, but what if we have a Hungary 2021 race too? Then the city would get duplicated.

3.5 Third Normal Form

Definition

"To be in **third normal form**, the relation must be in **second normal form**. Also, all **transitive dependencies** must be **removed**; a non-key attribute may not be functionally dependent on another non-key attribute." (Watt & Eng, 2014)

All three the tables `Driver`, `Race` and `DriverRace` are already in 2NF. Now we just need to remove any remaining transitive dependencies.

3.5.1 Step 6: Remove Any Transitive Dependencies

We have already identified the last remaining transitive dependency:

`Racetrack → Racetrack City`

To do this, we need to create another new entity with these two fields. The determinant here will become the PK of the new table.
`Racetrack (Racetrack, RacetrackCity)`

And then we can use that `Racetrack` PK as an FK in `Race`. Note that the FK is not underlined here since it doesn't form part of the PK.

`Race (Race, Racetrack)`

Driver and DriveRace remain unchanged since they didn't have any transitive dependencies to start with.

```
Driver (Driver, DriverCountry, DriverDoB)
DriverRace (Driver, Race, Position)
```

3.5.2 Step 7: Draw a New Dependency Diagram

Now we can draw the last dependency diagram, showing the four tables that we have now created with no remaining partial or transitive dependencies.

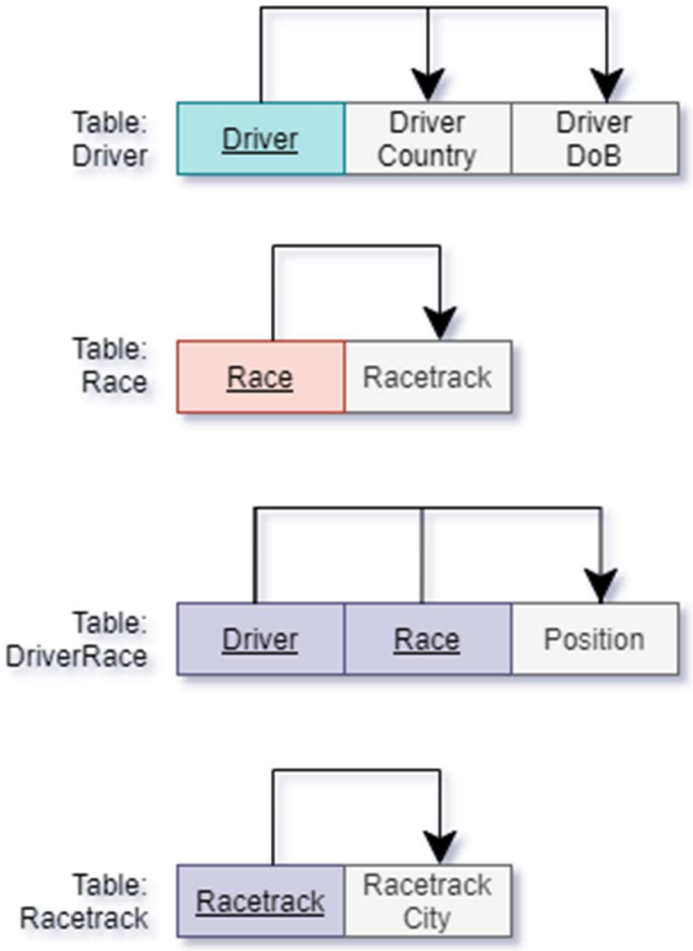


Figure 53. 3NF Dependency Diagram

3.6 Further Improvements

We started the normalisation process by saying that we must work with a composite natural PK for the process to work. But now that we have arrived at 3NF, we do not need to stick to that anymore. Now we can make improvements on the model, like adding a surrogate primary key for each table.

And if we do that, we see that we get to something that looks a lot like what we had in the ERD. The data that we worked with didn't include the team that the driver raced for in a specific race, so the ERD is more complete in that regard. But the normalised data added some additional attributes that we didn't know about before: `RacetrackCity` and `DriverCountry` and `DriverDoB`. So, we can add those to the ERD.

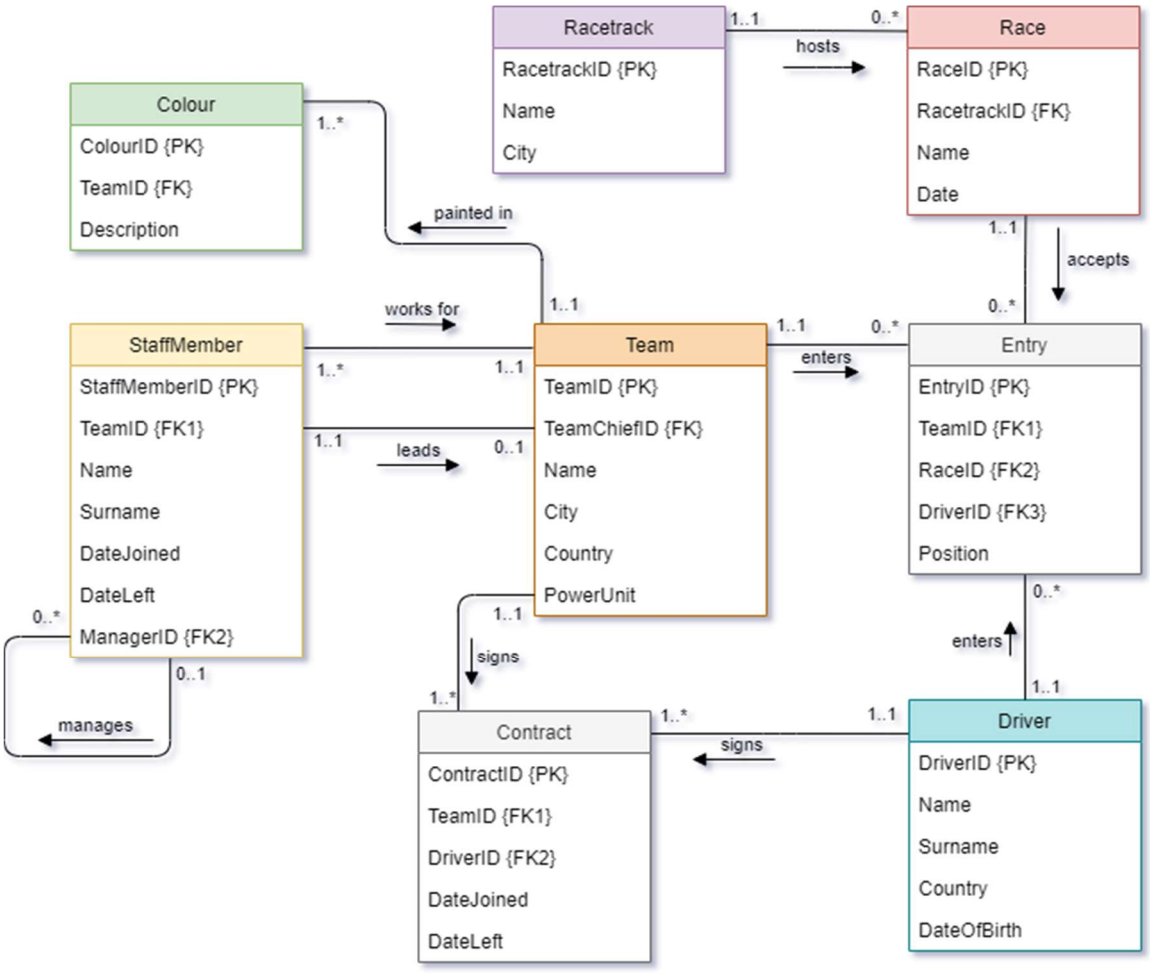


Figure 54. ERD with New Attributes Added

This illustrates why ERDs, and normalisation should go hand in hand. Neither process necessarily has all the answers but used together we can arrive at a great model!

4 Data Dictionaries

4.1 What is a Data Dictionary?

We now have a very good idea of what we want to store in the database for the Formula 1 racing data. We have defined all the entities, their keys, and other attributes. But there are more details that are required before we can implement the database.

In a strongly typed language like Java or C#, it is necessary to specify the datatype of each variable. For example, we could have: `int age;`

That datatype (`int` in this case) tells the compiler what you want to store, and how much memory to allocate for that variable. In both Java and C#, an `int` takes up 4 bytes, and it is a signed value (can be negative).

In the same way, when we create a **table** in a **relational database**, we need to specify what the **datatype** is for each column. And that will determine how much disk space it will take up.

The information that we will include in a data dictionary are (Kalodikis, 2016):

- Field name
- Data type
- Data format
- Field size
- Required (Knight, 2017)
- Description
- Example

Data dictionaries are sometimes considered to be an old-fashioned way of describing data. But this information will be invaluable when we start implementing the database in the next learning unit.

Side note: Once the database is implemented, and all the tables have been created, our DBMS will use an internal data dictionary to store the metadata. MySQL has a transactional data storage mechanism for storing this meta data. (Oracle Corporation, n.d.)

4.2 Creating a Data Dictionary

For the purposes of documenting this most detailed level of design, we are just going to create a table in Microsoft Excel or even Microsoft Word. The thinking behind choosing datatypes is what is important here, more than the tool that we use to document this.

Let us look at the `StaffMember` entity as an example.

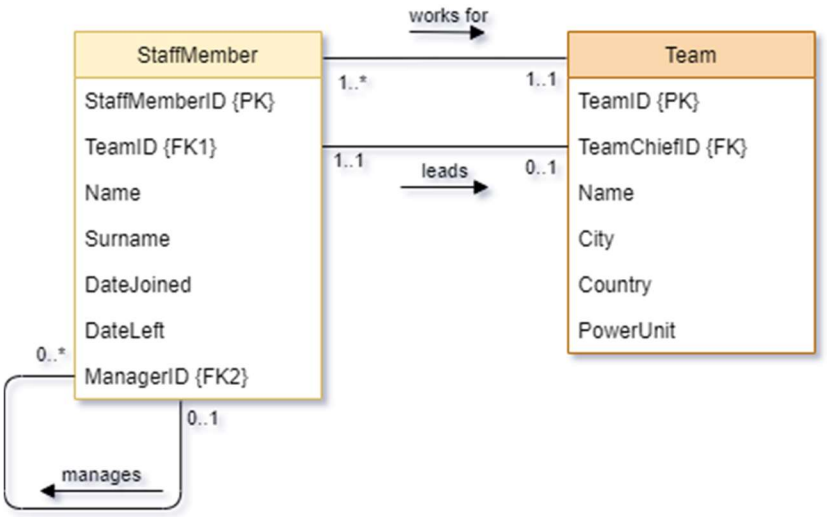


Figure 55. StaffMember Entity with Relationships

Figure 55 shows the `StaffMember` entity together with its relationships. We will see that these relationships are important when determining whether a field is required or not.

We start by creating a table that lists all the fields. Attributes in the ERD context are called fields when we talk about database tables.

Table: StaffMember

Field name	Data type	Data format	Field size	Req?	Description	Example
StaffMemberID {PK}						
TeamID {FK1}						
Name						
Surname						
DateJoined						
DateLeft						
ManagerID {FK2}						

Figure 56. Data Dictionary without Details

Then we populate all the information into the data dictionary.

`StaffMemberID` is the surrogate primary key of this table. It is good practice to make the surrogate PK an **integer** since it uses far less space than a universally unique identifier (UUID), which is typically 16 bytes. And if we follow that same logic, we know that all the surrogate keys will also be integers. When an FK is created, the FK column must be the same datatype as the PK of the other table. So, the FKs also all need to be integers.

For integers PKs and FKs, we don't need to specify the data format and field size – these are just going to be numbers that are used in the background. But we can write a description and an example.

Foreign keys, by their definition, need to either be a valid primary key in the other table or null. If it is null, it means that there is no row in the other table that is related. Looking at the relationship between `StaffMember` and `Team` in Figure 55, we see that each `StaffMember` works for exactly one `Team` – the minimum is 1. That means that the TeamID FK is required and is never allowed to be null. It makes sense if you think about it. Why would we store somebody in this database if they don't belong to a team?

Let us consider the other FK: `ManagerID`. Each `StaffMember` is managed by zero or one other `StaffMembers`. The fact that it can be zero (remember, the top-level people don't have managers in this database), means that it is not a required field. It is thus totally a valid thing to do to create a `StaffMember` with a null value for `ManagerID`.

It is important to note that if we make a field mandatory in the database, the DBMS will not allow us to violate that constraint. Besides those hard constraints, we could also implement validations in the software that makes use of the database. Those checks could implement more specific things; for example the data joined cannot be earlier than the first race that was ever held.

Let us fill those details into the data dictionary.

Table: StaffMember

Field name	Data type	Data format	Field size	Req?	Description	Example
StaffMemberID {PK}	Integer			Yes	The primary key field that uniquely identifies the StaffMember	1
TeamID {FK1}	Integer			Yes	The foreign key of the team that the StaffMember works for	1
Name						
Surname						
DateJoined						
DateLeft						
ManagerID {FK2}	Integer			Yes	The foreign key of the StaffMember that manages this StaffMember	1

Figure 57. Data Dictionary with Key Details

Next, we have two similar fields: name and surname of the driver. Both are text fields that need to be long enough to contain the name or surname of any driver. Here we can specify the field size, to be a maximum of 50 characters. These two fields are both required.

The example name and surname illustrate something important here – names and surnames can have special characters. When we get to the implementation details in the next learning unit, we will see that there are different ways to represent a text field. But for now, it is worth noting that the example values should not stick to the simple values – try to find the more complex examples.

Lastly, we have the dates joined and left. `DateJoined` is mandatory, but the rules say that `DateLeft` is optional. In the case of dates and times, the data format is particularly useful. Here we can capture that we want the date in the format DD/MM/YYYY.

Now we have the whole data dictionary created for the `StaffMember` table – see Figure 58.

Table: StaffMember

Field name	Data type	Data format	Field size	Req?	Description	Example
StaffMemberID {PK}	Integer			Yes	The primary key field that uniquely identifies the StaffMember	1
TeamID {FK1}	Integer			Yes	The foreign key of the team that the StaffMember works for	1
Name	Text		50	Yes	The first name of the StaffMember	François
Surname	Text		50	Yes	The surname of the StaffMember	Räikkönen
DateJoined	Date	YYYY-MM-DD		Yes	Date that the StaffMember joined the Team	2019-01-14
DateLeft	Date	YYYY-MM-DD		No	Date that the StaffMember left the Team	2020-05-06
ManagerID {FK2}	Integer			Yes	The foreign key of the StaffMember that manages this StaffMember	1

Figure 58. Data Dictionary with All Details

5 Data Security

5.1 Designing for Security

Security in information technology is a huge area of study. And for good reason – companies and individuals want their sensitive information to be protected. There are many steps that can be taken to keep databases in general, safe from intrusion. For example, by limiting the access that authorised users have to data that is specific to what they need to perform their jobs. (Howarth, 2014)

For our purposes in this module though, let's limit the discussion to the data that specifically needs secure storage.

5.1.1 Passwords

The most sensitive piece of information that is widely stored is a **password**. By its very nature, if the password were compromised, this could allow unauthorised third parties to access a user's account.

For this reason, **never ever store passwords in clear text in a database**, no matter how secure the database itself is perceived to be. Instead, store a hash of the password.

Definition

"**Hashing** performs a **one-way transformation** on a password, turning the password into another String, called the hashed password. 'One-way' means that it is practically impossible to go the other way – to turn the hashed password back into the original password." (Oracle, n.d.)

Several hashing algorithms have been created over the years. And not all of them are considered cryptographically secure enough to use for passwords. Read (Kauffman, 2013) for more information.

5.1.2 Personal Information

The protection of personal information is getting written into law in South Africa in the **Protection of Personal Information** (**PoPI**) **Act** (Act no 4 of 2013). Let's look at the definition according of personal information to this Act.

"'**personal information**' means information relating to an identifiable, living, natural person, and where it is applicable, an identifiable, existing juristic person, including, but not limited to:

a) information relating to the race, gender, sex, pregnancy, marital status, national, ethnic or social origin, colour, sexual orientation, age, physical or mental health, well-being, disability, religion, conscience, belief, culture, language and birth of the person;

b) information relating to the education or the medical, financial, criminal or employment history of the person;

c) any identifying number, symbol, e-mail address, physical address, telephone number, location information, online identifier or other particular assignment to the person;

d) the biometric information of the person;

e) the personal opinions, views or preferences of the person;

f) correspondence sent by the person that is implicitly or explicitly of a private or confidential nature or further correspondence that would reveal the contents of the original correspondence;

g) the views or opinions of another individual about the person; and

h) the name of the person if it appears with other personal information relating to the person or if the disclosure of the name itself would reveal information about the person." (South African Government, 2013)

Under the PoPI Act, companies are required to handle all this personally identifiable data with special care. It means that data like this cannot be stored unsecured, especially not outside the geographical boundaries of South Africa. It also means that users need to provide consent for their data to be stored and used.

There is a whole site dedicated to information about how to comply with the PoPI Act. Read more in (www.popiact-compliance.co.za, n.d.).

6 Recommended Additional Reading

Coronel, C., Morris, S., Crockett, K. and Rob, P. 2013. Chapters 5 and 7 in *Database Principles: Fundamentals of Design, Implementation and Management*. 2nd ed. Cengage Learning EMEA.

Dybka, P., 2014b. *Chen Notation.* [Online] Available at: https://www.vertabelo.com/blog/chen-erd-notation/ [Accessed 11 December 2023].

Dybka, P., 2016. *Crow's Foot Notation.* [Online] Available at: https://www.vertabelo.com/blog/crow-s-foot-notation/ [Accessed 11 December 2023].

Dybka, P., 2014c. *UML Notation.* [Online] Available at: https://www.vertabelo.com/blog/uml-notation/ [Accessed 11 December 2023].

Larsen, G. A., 2011. *SQL Server: Natural Key Verses Surrogate Key*. [Online] Available at: https://www.databasejournal.com/features/mssql/article.php/3922066/SQL-Server-Natural-Key-Verses-Surrogate-Key.htm [Accessed 11 December 2023].

Monge, A. n.d. *Database design with UML and SQL, 4th edition*. [Online] Available at: https://web.csulb.edu/colleges/coe/cecs/ dbdesign/dbdesign.php [Accessed 11 December 2023].

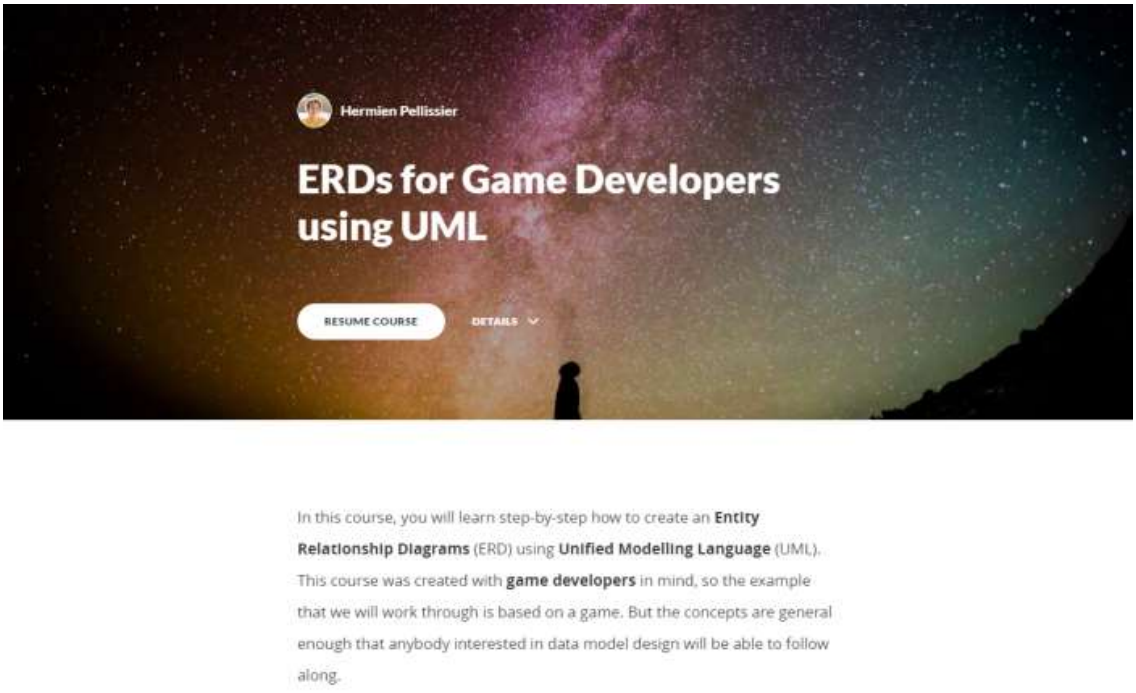
Sezairi, A., 2019. *What is an Entity-Relationship Diagram?.* [Online] Available at: https://medium.com/better-programming/what-is-an-entity-relationship-diagram-d5db69a87971 [Accessed 11 December 2023].

University of Cape Town, n.d. *Chapter 12. Database Security.* [Online] Available at: https://www.cs.uct.ac.za/mit_notes/ database/pdfs/chp12.pdf [Accessed 11 December 2023].

Watt, A. and N. Eng. 2014. Chapter 8 in *Database Design.* 2nd edition. Victoria, B.C. Available at: https://open.bccampus.ca/ browse-our-collection/find-open-textbooks/?uuid=5b6f010a-0563-44d4-94c5-67caa515d2c5 [Accessed 11 December 2023].

7 Recommended Digital Engagement and Activities

Do the below free course that works step-by-step through drawing UML ERDs for a game development-themed example.



Pellissier, H. 2019. *ERDs for Game Developers using UML*. [Online] Available at: https://www.pellissier.co.za/wp-content/uploads/erds/content/index.html#/ [Accessed 11 December 2023].

8 Revision Exercises

8.1 Revision Exercise 1

Why should ERDs and normalisation be used in conjunction with one another when designing a database?

8.2 Revision Exercise 2

Draw an entity relationship diagram (ERD) in Unified Modelling Language (UML) notation according to the following banking system business rules:

1. All entities must have a surrogate primary key.
2. Each branch of a bank has a name and a branch code that must be stored in the database.
3. Each client is assigned exactly one branch, which is their home branch. And each branch can have many clients.
4. An account belongs to a specific client and is opened at a specific branch. Clients and branches can have many accounts.
5. An account has a description, account number, and date opened that must be stored in the database.
6. The name, surname, and ID number for each client must be stored in the database.

8.3 Revision Exercise 3

The below data is already in its first normal form. (The names and ID numbers below are all fictional.) Normalise it to the third normal form (3NF):

Primary key: Branch ID, Client ID

Branch ID	**Branch Name**	**Client ID**	**Client Name**	**Client Surname**	**Client ID Number**
1	Joburg	1	Bob	McCartney	5101010000082
1	Joburg	2	Sarah	Smith	8212120000089
2	Cape Town	3	Thabo	Ntshinga	7303130000054

9 Solutions to Revision Exercises

9.1 Revision Exercise 1

Read Section 3.

9.2 Revision Exercise 2

Read Section 2.

9.3 Revision Exercise 3

Read Section 3.

Learning Unit 3: Creating a Relational Database	
Learning Objectives: • Explain the difference between SQL and programming languages. • Create schemas. • Create tables. • Create tables with constraints. • Use SQL to insert data into a database. • Use SQL to update data. • Use SQL to delete data. • Explain the purpose of transactions.	**My notes**
Material used for this learning unit: • This module manual. • MySQL server installation.	
How to prepare for this learning unit: • Read through the content in this learning unit.	

1 Introduction

In Learning Unit 2, we looked at all the aspects of designing a database. Now we are ready to start implementing a relational database.

Figure 59. Basic Data Persistence Operations

Figure 59 shows the four basic operations that can be done when persisting data. In this learning unit, we are going to look at how to create tables, how to insert rows in those tables, how to update data, and how to delete data.

In Learning Units 4 and 5, we will focus more on the reading of data in various ways.

2 Introduction to SQL

2.1 What is SQL?

In Learning Unit 1, we already saw that Structured Query Language (SQL) is a declarative language. Using this type of language, a developer can specify what needs to be done without having to get into the nitty-gritty details of how that is accomplished.

When reading about database languages, you will come across two concepts: **Data Definition Language** (**DDL**) and **Data Manipulation Language** (**DML**). A DDL is used to create users, tables, and so on, which a DML is used to create and read the data. (Parahar, 2019) In modern databases, SQL is used for both of these aspects, making the distinction less important. But it does show what can be done using SQL, and it helps to find information in the MySQL documentation.

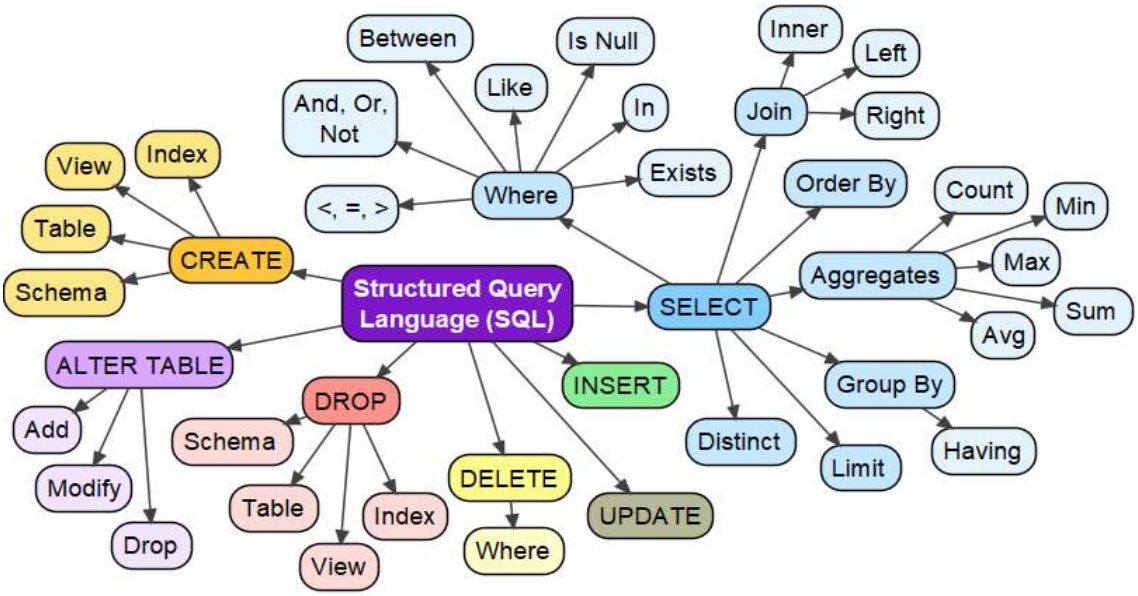


Figure 60. Overview of SQL Statements

In Figure 60, we see an overview of all the most important SQL statements that we are going to learn more about. Looking at the MySQL 8.0 Reference Manual (Oracle Corporation, n.d.b) – the definitive source of information about the MySQL dialect of SQL – we see that the following are examples of data definition statements:

- ALTER
- CREATE
- DROP

And the following are examples of data manipulation statements:

- INSERT
- SELECT
- UPDATE

What is a **SQL dialect**? Each vendor that has a SQL database implements SQL. But there are slight variations in the details of exactly how it is implemented. It is like the difference between South African English and American English – there are differences, but we can understand American English speakers.

Similarly, the differences between the SQL dialects are also mostly subtle. For example, data in MySQL is not case sensitive while data in Microsoft SQL Server is case sensitive. Also, the functions that are used for working with dates are different. (Khalil, 2018)

Read (Khalil, 2018) for more excellent examples of the differences between the dialects.

We are going to use MySQL throughout the SQL learning units of this module. So, when in doubt, stick to the MySQL syntax.

Tip: When searching online for SQL information, make sure to include MySQL in your search terms!

When reading the official MySQL documentation about SQL statements, there are some documentation conventions that we need to know about. These are shown in Figure 61.

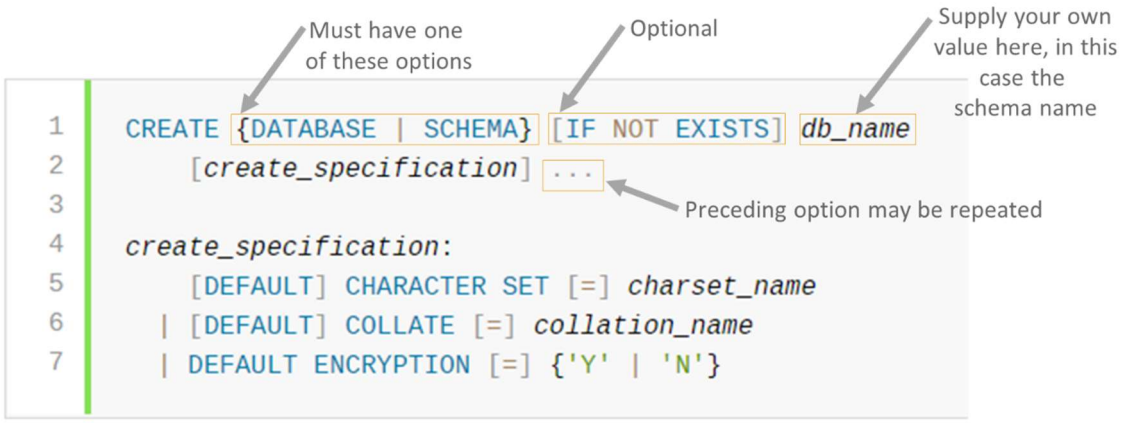


Figure 61. MySQL Documentation Conventions (From (Oracle Corporation, n.d. c) with annotations)

When writing SQL code, just like Java and C# code, it is good practice to be consistent and stick to the conventions of the SQL dialect. For example, write the keywords like CREATE and DATABASE in all caps.

2.2 Installing MySQL

If you want to install MySQL on your own computer, make sure that you download the MySQL Community Server edition, available from https://dev.mysql.com/downloads/mysql/. This is the open-source version of the server, and can be downloaded without logging in with an Oracle account. (The Enterprise edition of the server is a commercial offering that provides advanced features that we won't need for this module.)

The Community Server is available for different operating systems, including Windows and several different Linux flavours. Detailed installation instructions can be found on the below pages, depending on your operating system:

- Windows: https://dev.mysql.com/doc/mysql-installation-excerpt/8.0/en/windows-installation.html [Accessed 11 December 2023].
- Linux: https://dev.mysql.com/doc/mysql-installation-excerpt/8.0/en/linux-installation.html [Accessed 11 December 2023].
- Other operating systems: https://dev.mysql.com/doc/mysql-installation-excerpt/8.0/en/ 3[Accessed 11 December 2022].

During the installation, you will be prompted to select which components to install. The only components that we will need for this module are **MySQL Server** and **MySQL Workbench**. If you want to save bandwidth, there is no need to download the rest of the components.

During installation, you will be asked to provide a password for the admin account. Make sure that you **remember** what that **password** is – you will need it to connect to the database later!

2.3 Using MySQL Workbench

MySQL Workbench is the easiest way to interact with the local instance of the database. Click **Local instance** to connect to the local database.

Figure 62. Connecting to the Local Instance

The default port that the MySQL database is installed on is 3306.

The main window of the MySQL Workbench application will appear, as shown in Figure 63. This instance of the DBMS already has a schema called `gameresources` that was created, and another called `stamp_database`. These existing schemas and their contents can be browsed using the **Navigator** window. If you don't have any databases yet, you will only see `sys` – a system database that contains configuration properties.

Note that the currently **active database** is shown in **bold** in the Navigator window. Any queries that are executed will, by default, be executed against that database.

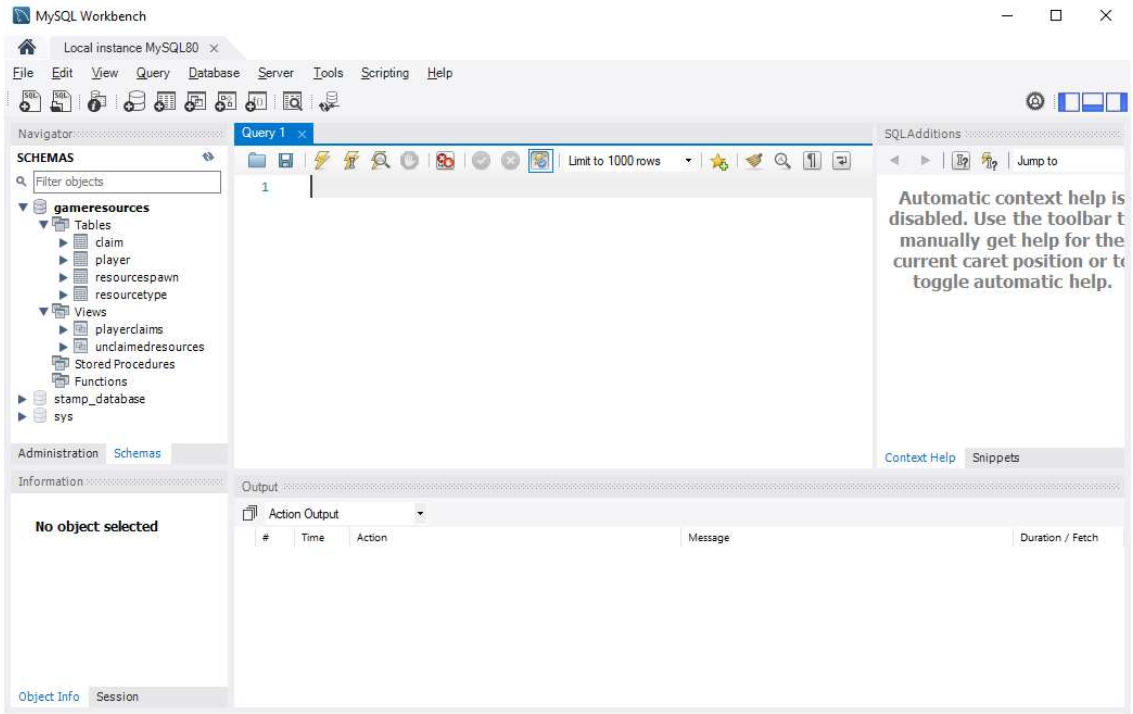


Figure 63. Main Window of the MySQL Workbench

In the editor area, there are various things that can be done using the graphical user interface (GUI). For this module, we will be writing our own scripts instead of relying on the GUI tools, so click the first button on the toolbar (**Create a new SQL tab for executing queries**) to create a blank SQL script. That script can be saved with a .sql extension, but it is just a plain text file that can be viewed and edited with any text editor.

The SQL statements in the current SQL script can be executed by clicking the button with the lightning bolt icon just above the script.

3 Creating a Single Table

In this section, we are going to look at how to create a table that doesn't have any foreign keys. Related tables will be in the next section.

3.1 Working with Schemas

We are again going to work with the Formula 1 data here. And the first thing that we want to do is create a new schema, which is going to contain all the tables and data that we want to create.

Before we do that, let us look at case sensitivity of names (for example, for tables) in MySQL. Is MySQL **case sensitive**? It depends on the operating system that the server runs on and the settings of the server. By default, table names, etc., are stored in lower case format on Windows but are not case sensitive. (Oracle Corporation, n.d. d)

What this means is that on Windows, if you create a schema called Formula1Racing, the name will by default be converted to formula1racing. It is useful to know about this, so we can decide to add underscores (_) to improve the readability of our names.

Back to creating our schema. Figure 64 shows the SQL statements used to create our new schema and set it as the default database that we are going to use going forward.

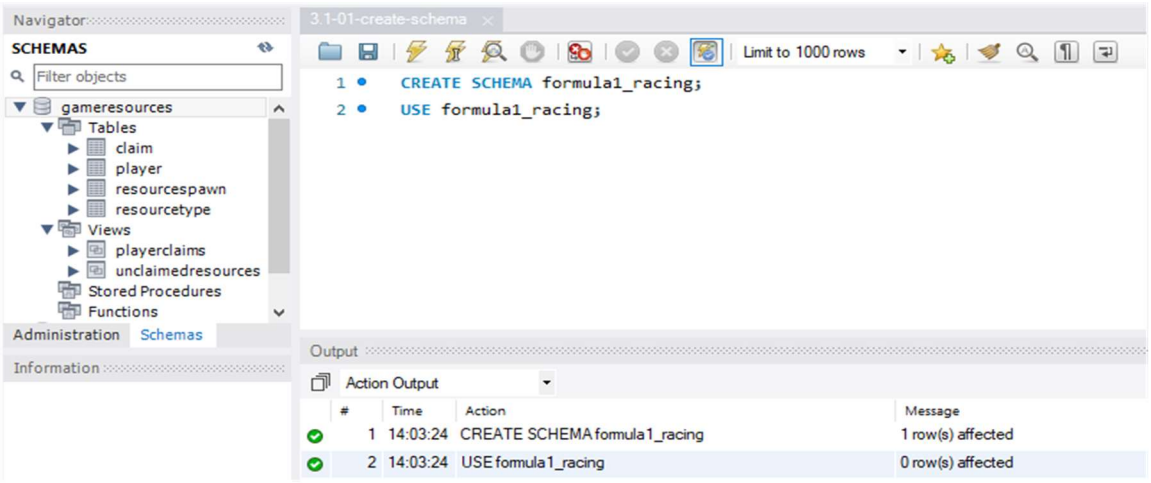


Figure 64. CREATE SCHEMA and USE

After running these two statements, the output window shows us what happened. When the schema was created, one row was affected. What happened behind the scenes is that the schema was added to the data dictionary that the MySQL DBMS uses to store metadata.

Notice that there is a semicolon at the end of the line for each of these statements. The semicolon is optional if there is only one statement in the file. But if there are multiple statements, all of them except the last one must have semicolons.

Looking carefully at the **Navigator** window, we see that the new schema does not appear yet. And the reason is that you need to refresh the list manually by clicking on the small double arrow button next to **SCHEMAS**. This is a good optimisation – if MySQL were refreshing this all the time, it would generate a lot of unnecessary database traffic. But do remember to refresh if you do not see what you expect to see.

If we were to run the `CREATE SCHEMA` statement a second time, we would see an error message in the output window as shown in Figure 65. And that means that the script stops executing at that point.

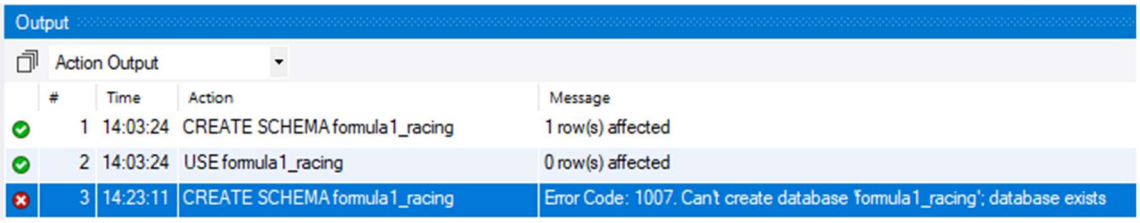


Figure 65. Database Exists Error Message

The new log messages get appended to the end of the output view until the user decides to clear the list. Notice how the `USE` statement did not appear in the log again – execution stopped at the error.

But what if we wanted to create the database if it does not exist yet, and just continue with the rest of the script if it does exist? We can add `IF NOT EXISTS` to the `CREATE SCHEMA` statement. Notice that now the message is only a warning in Figure 66, and the script continues to run afterwards.

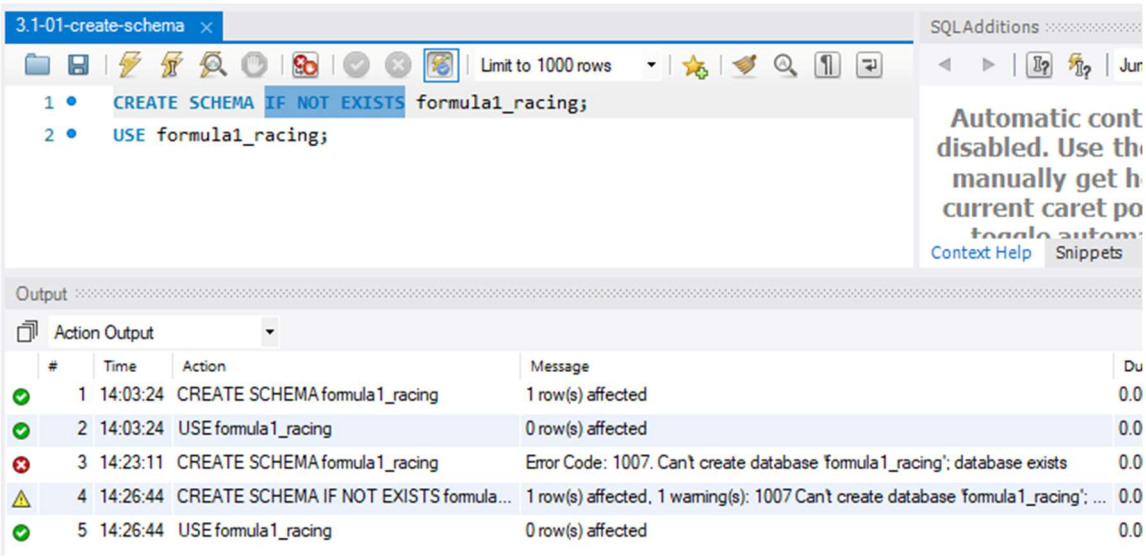


Figure 66. Database Exists Warning Message

Look again at Figure 61 (the one about the documentation conventions – see p.86). This is a good example of an optional statement that can be added if necessary. The same can also be done when creating tables.

3.2 Creating a Table

We are going to create the table `Racetrack`. The entity is shown in Figure 67 for reference.

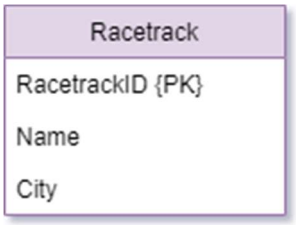


Figure 67. Racetrack

Let us quickly create the `Racetrack` data dictionary too, so we can have all the information that we need to create the table.

The name of the racetrack can potentially be quite long since it often does include the name of the sponsor. So, the file size for that needs to be larger.

Table: Racetrack

Field name	Data type	Data format	Field size	Req?	Description	Example
RacetrackID {PK}	Integer			Yes	The PK that identifies the Racetrack.	1
Name	Text		250	Yes	The name of the Racetrack.	Red Bull Ring
City	Text		50	Yes	The city in which the Racetrack is located.	Spielberg

Figure 68. Data Dictionary for Racetrack

In the data dictionary, we just specified so far that the `Name` and `City` will contain text. But MySQL has different types that are more specific that can be used here (Oracle Corporation, n.d. e):

- CHAR(n) – Fixed length string of length n, padded with spaces at the end. Maximum length of 255.
- VARCHAR(n) – Variable length string of length n, that is not padded with spaces but does have a length prepended to the string. Maximum length of 65535.

In each case, the character set that can be used can also be specified. That will determine the letters that can be stored and will also change the number of bytes used to represent each letter.

Character sets include `ascii`, `latin1` and `utf8mb4` (UTF-8 Unicode). (Oracle Corporation, n.d. f)

Here we are going to use `utf8mb4` since that provides us with the ability to store lots of special characters. But very few of the names will be 250 characters long, so we are going to use `varchar` for this.

```
1  CREATE TABLE racetrack (
2      racetrack_id    INT AUTO_INCREMENT NOT NULL,
3      name            VARCHAR(250) CHARACTER SET utf8mb4 NOT NULL,
4      city            VARCHAR(50) CHARACTER SET utf8mb4 NOT NULL,
5      PRIMARY KEY (racetrack_id)
6  );
```

Figure 69. CREATE TABLE for a single table

We start with the statement `CREATE TABLE` followed by the name of the table. Since MySQL is going to convert it to lowercase by default on Windows anyway, we might as well keep lower case names.

Notice that the brackets here are round brackets () and not curly braces {} as you are used to in Java or C# programming.

Put each field on a separate line to make the code more readable. And notice that there is a comma at the end of each line here. Without that, the SQL statement will not run.

By making use of the `NOT NULL` statement, we tell the database that nulls are not allowed. This means that a value for the field is always required, as we indicated in the data dictionary.

The `racetrack_id`, which is the PK for this table, is declared here to be `AUTO_INCREMENT`. This means that a sequential value will automatically be assigned to each new row that is added. This is great for a PK – we don't need to worry about the values being unique.

The last line adds the **primary key constraint** to the table. If we had a composite PK, multiple columns could be specified in the same way.

Once this SQL statement has been executed, the table now exists in the database. But there is no data in it yet.

3.3 Altering a Table

One way of changing the structure of a table would be to drop it (completely delete the whole table and all its data) and recreate it. That is fine right now since we do not have any data yet. But if this database were in a production system and contained all customer records already, that would be a very destructive way to go about changing a table's structure.

Surely, though, tables do not stay the same forever. What if business requirements change and you must add a column to an existing table? `ALTER TABLE` to the rescue!

```
1 •  ALTER TABLE racetrack
2    ADD COLUMN country VARCHAR(50) CHARACTER SET utf8mb4 NOT NULL;
3
4 •  ALTER TABLE racetrack
5    CHANGE COLUMN country countryNew VARCHAR(100) CHARACTER SET utf8mb4 NOT NULL;
6
7 •  ALTER TABLE racetrack
8    DROP COLUMN countryNew;
```

Figure 70. ALTER TABLE Statements

Figure 70 shows some of the most common use cases of `ALTER TABLE`. It can be used to add a column, change the name and/or definition of an existing column, and even drop (permanently delete) a whole column. Dropping a column is permanent, and the data really will be gone forever, so use with care!

Pro tip: Microsoft SQL Server has the concept of a DACPAC that can be used to safely make changes to a database.

3.4 Dropping a Table or Schema

A table or schema can be permanently deleted, including all the data that is contained in them, by using the SQL statement `DROP`.

```
1 •   DROP TABLE racetrack;
2 •   DROP SCHEMA formula1_racing;
```

Figure 71. DROP Statements

`DROP` can be very useful in a testing environment, where, for example, unit tests need to be run against a clean database. `DROP` the whole database and create it afresh using SQL scripts, then you know that the tests don't break because the database is out of date.

These `DROP` statements are destructive and should be used with the utmost case in production databases!

4 Creating Related Tables

4.1 Foreign Keys and Constraints

So far, so good. We can now create tables that stand on their own. But what about related tables?

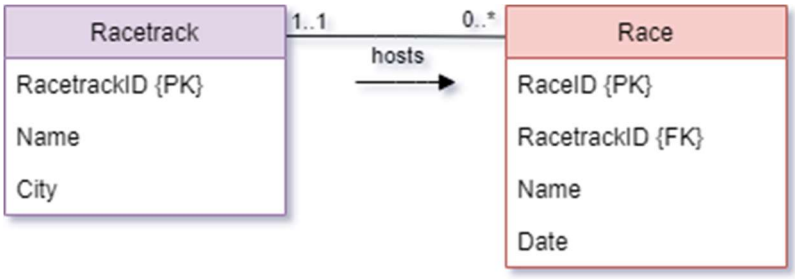


Figure 72. Related table: Race

Figure 72 shows the next table that we want to create: `Race`. This table has a single FK that relates it to `Racetrack`. And we see from the diagram that each `Race` must be hosted by exactly one `Racetrack`, so that field is mandatory.

Table: Race

Field name	Data type	Data format	Field size	Req ?	Description	Example
RaceID {PK}	Integer			Yes	The PK that identifies the Race.	1
RacetrackID {FK}	Integer			Yes	The FK that identifies the Racetrack where the Race takes place.	1
Name	Text		250	Yes	The name of the Race.	Austria 2020
Date	Date	YYYY-MM-DD		Yes	The date on which the Race takes place.	2020-06-30

Figure 73. Data Dictionary for Race

The data dictionary for `Race` is shown in Figure 73. Here we have a new data type that we have not encountered yet: date. MySQL has a `DATE` type, which will work well here. The format that MySQL stores and retrieves dates in is YYYY-MM-DD. (Oracle Corporation, n.d. g)

For the name, we will again use a `VARCHAR` with the character set utf8mb4.

Now we have all the information at hand to create the `Race` table.

```
1  CREATE TABLE race (
2      race_id         INT AUTO_INCREMENT NOT NULL,
3      racetrack_id    INT NOT NULL,
4      name            VARCHAR(250) CHARACTER SET utf8mb4 NOT NULL,
5      date            DATE NOT NULL,
6      PRIMARY KEY (race_id),
7      FOREIGN KEY (racetrack_id) REFERENCES racetrack(racetrack_id)
8  );
```

Figure 74. CREATE TABLE with foreign key constraint

Notice that we first create the column `racetrack_id` in this table, and then only do we add the foreign key constraint that links it to the `Race` table.

With the table created like this, the database will enforce the fact that it is a foreign key. So, it will not allow invalid values (that are not valid PKs in the `Racetrack` table) to be inserted.

5 Inserting, Updating and Deleting Data

5.1 Inserting Data into a Single Table

A database with a bunch of empty tables is not particularly useful. So, let's see how to insert some rows into our `Racetrack` table.

```
1 •  INSERT INTO racetrack (name, city)
2    VALUES ("Red Bull Ring", "Spielberg"),
3           ("Hungaroring", "Budapest");
4
5 •  SELECT *
6    FROM racetrack;
```

Result Grid | Filter Rows: | Edit:

racetrack_id	name	city
1	Red Bull Ring	Spielberg
2	Hungaroring	Budapest
NULL	NULL	NULL

Figure 75. INSERT INTO

The PK field was created to be an auto-number, so we don't need to provide a value for that. It will be created automatically. So, we only need to specify the values for the `name` and `city` fields. And we can insert multiple rows at the same time, as shown in Figure 75. Notice that the string values are specified in double quotes. If we left out the quotes, the query would not run because then those would not be literal string values.

Also, in Figure 75, we see our first `SELECT` statement. A `SELECT` statement is used to retrieve data from the database. Here we just get everything from the table `Racetrack` so we can see the effect that the `INSERT INTO` had. We will look at `SELECT` statements in much more detail in the next learning unit.

5.2 Inserting Data into Related Tables

Now we also need to create the races that are taking place at the various racetracks. For that, we need to specify the ID of the Racetrack, the `name` of the race and the `date` on which it takes place.

```
1 •  INSERT INTO race (racetrack_id, name, date)
2    VALUES (1, "Austria 2020", "2020-07-05"),
3           (2, "Hungary 2020", "2020-07-19");
4
5 •  SELECT *
6    FROM race;
```

Result Grid | Filter Rows: | Edit:

race_id	racetrack_id	name	date
1	1	Austria 2020	2020-07-05
2	2	Hungary 2020	2020-07-19
NULL	NULL	NULL	NULL

Figure 76. INSERT INTO Race

In this specific example, it happens that the first race, Austria 2020, takes place at Racetrack number 1. But that is entirely coincidental. The important point is that we must specify a value for the FK since we declared it mandatory (not null) when we created the table. If we tried to run the query without a value for the FK, we would get an error (see Figure 77).

Figure 77. Field doesn't have a value

5.3 Updating Data

Some data in a database should not be updated (or deleted, for that matter). This is particularly true for financial data. If a transaction happened, you cannot just modify or delete it. Accountants and auditors will tell you that to reverse a transaction you need to do the opposite transaction, instead of deleting it. Or for a partial refund, create a new entry for that refund value.

However, there are cases where it should be possible to update data. What if the schedule for the races changes, for example, and the date now needs to be updated?

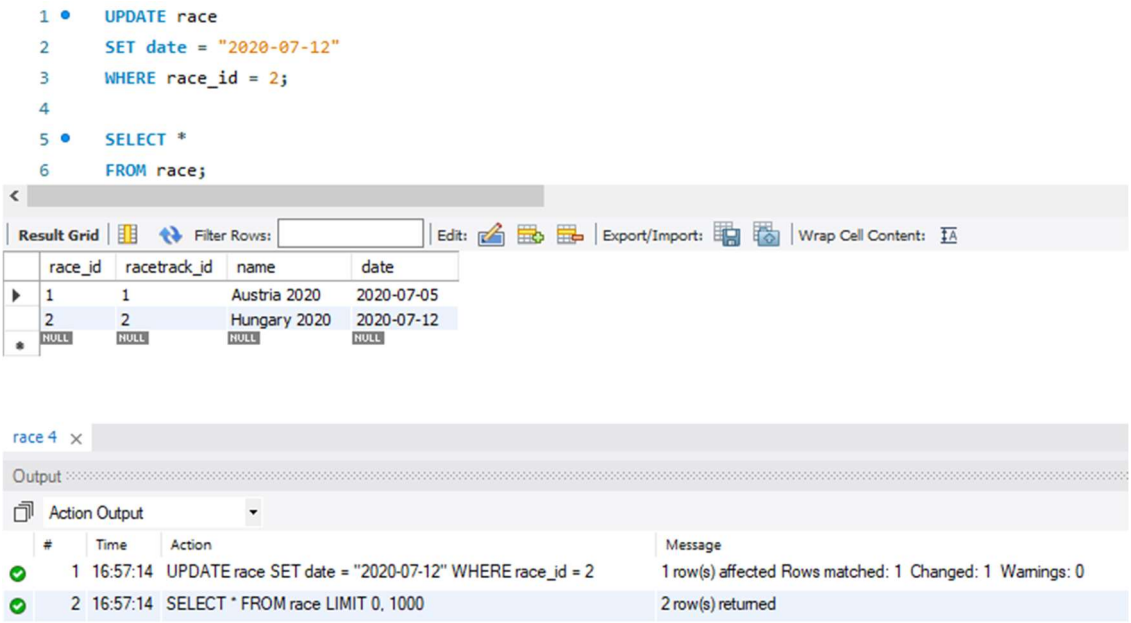


Figure 78. UPDATE with WHERE clause

In Figure 78, we see an `UPDATE` statement that updates entries in the `Race` table. Without the `WHERE` clause, all the rows would be updated. But the `WHERE` clause limits the change to a single row: the one where the `race_id` is two.

Be careful with `UPDATE` statements in production databases. If you forget to specify the `WHERE` clause, you can accidentally overwrite a lot of data that cannot be recovered!

Here we see the power of the fact that SQL is a declarative language. We simply tell the database to change the data for the second `Race`. There is no need to write a loop to find the entry, and then to update. The DBMS knows how to do this, and so it is enough for us to specify what we want it to do.

5.4 Deleting Data

We have already seen that whole tables can be dropped. But what if we wanted to delete only one specific row? There is a SQL statement for that!

In Figure 79 we see the `DELETE` statement that deletes one row (the one with `race_id` equal to 2) from the `Race` table.

Be careful with `DELETE` statements in production databases – the row(s) will be permanently deleted!

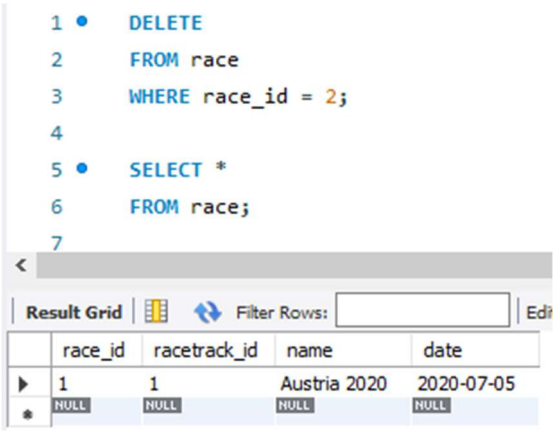


Figure 79. DELETE a single row

If you tried to delete a row that is referenced in a related table, the database will not allow that to happen. For example, we still have the race that takes place at `Racetrack` number 1. If we then tried to delete `Racetrack` number 1, the database would not allow it. The error message is shown in Figure 80.

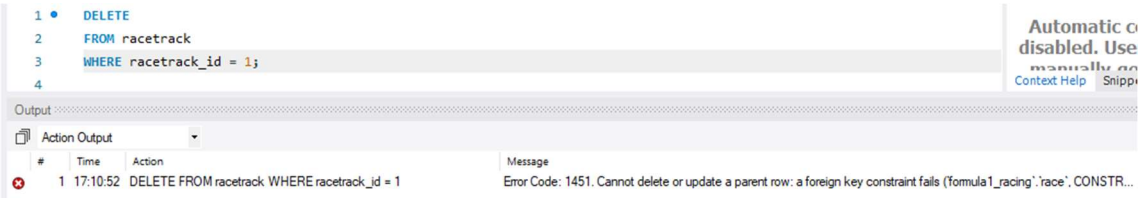


Figure 80. DELETE failed due to foreign key constraint

5.5 Transactions

We have used transactional and transactional a couple of times now. But what is a transaction in a database context?

Definition

"MySQL **transaction** allows you to execute a set of MySQL operations to ensure that the database never contains the result of partial operations." (mysqltutorial.org, n.d.)

Consider the transaction where money is transferred between two bank accounts. If the money is successfully deducted from the first account, but something goes wrong with transferring it to the other account, what should happen? The whole transaction should be reverted. Otherwise, the result would leave the database in an inconsistent state, and the money would be gone!

Such an error can be avoided by using a **transaction**.

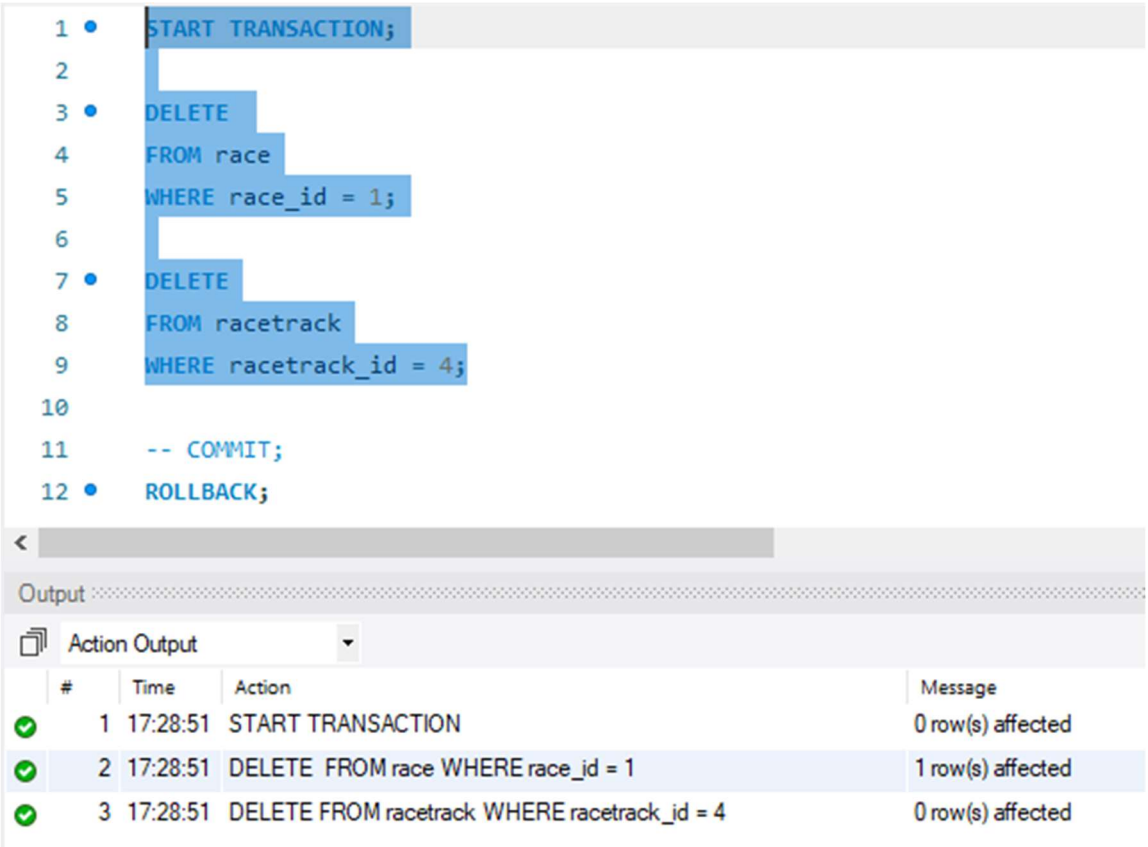


Figure 81. Transaction with DELETE statements

We start a transaction at the beginning of the SQL script shown in Figure 81. Then we delete a `Race` and try to delete the `Racetrack` too. But no `Racetrack` is deleted because we had the wrong ID here …

Now we have two options: `COMMIT` which will apply all the changes and stop the transaction. Or `ROLLBACK` which will undo all the changes since the transaction started and stop the transaction. In this case, we will `ROLLBACK`, fix the second `DELETE` statement, and try again.

Database transactions should be atomic, consistent, isolated, and durable (abbreviated as **ACID**) to properly maintain data integrity. **Atomic** means that a transaction should either occur completely or not at all. **Consistent** means that the data should be in a consistent state when the transaction ends. Transactions should be **isolated** from one another if multiple users make use of transactions at the same time. And **durable** means that the data should be written to persistent storage and should not be lost if there is a failure after the transaction completes. (Educative, Inc., 2020)

6 Recommended Additional Reading

MySQL Tutorial is a great website with a lot of resources about using MySQL. It is available here: https://www.mysqltutorial.org/ [Accessed 8 December 2022].

7 Recommended Digital Engagement and Activities

Khan Academy has some great content about SQL. Watch the **Welcome to SQL** video here:
https://www.khanacademy.org/computing/computer-programming/sql/sql-basics/v/welcome-to-sql [Accessed 11 December 2023].

And then do the **Creating a table and inserting data exercise**:
https://www.khanacademy.org/computing/computer-programming/sql/sql-basics/pt/creating-a-table-and-inserting-data [Accessed 11 December 2023].

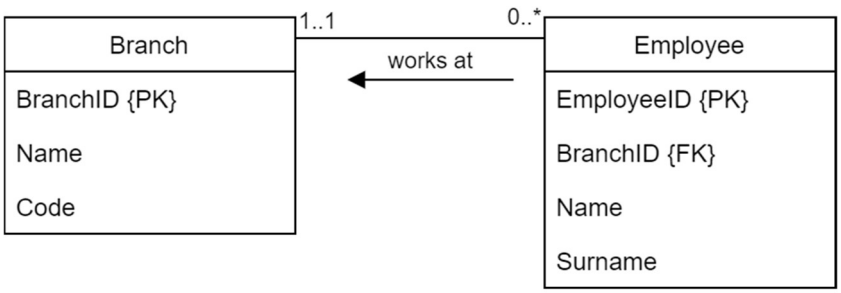
8 Revision Exercises

8.1 Revision Exercise 1

What is the difference between `DROP` and `DELETE`?

8.2 Revision Exercise 2

Given the below entity relationship diagram (ERD), write a SQL statement to create the employee table. Assume that the branch table already exists.



9 Solutions to Revision Exercises

9.1 Revision Exercise 1

Read Sections 3.4 and 5.4 above.

9.2 Revision Exercise 2

Read Section 4 above.

Learning Unit 4: Querying Data	
Learning Objectives: • Create queries to retrieve data from a database. • Explain how to order the results returned by a query. • Create queries that retrieve data from multiple tables. • Create queries that calculate aggregate values. • Use grouping to calculate aggregate values on groups of records.	**My notes**
Material used for this learning unit: • This module manual. • MySQL server installation.	
How to prepare for this learning unit: • Read through the content in this learning unit.	

1 Introduction

In Learning Unit 3, we learned how to create a schema with tables. And we inserted some data into the database and saw how to delete data. All of that is essential for working with a database, but if we cannot read the data out of the database again, then it is not useful at all. In this learning unit, we are going to learn how to write queries to get data out of the database in various ways.

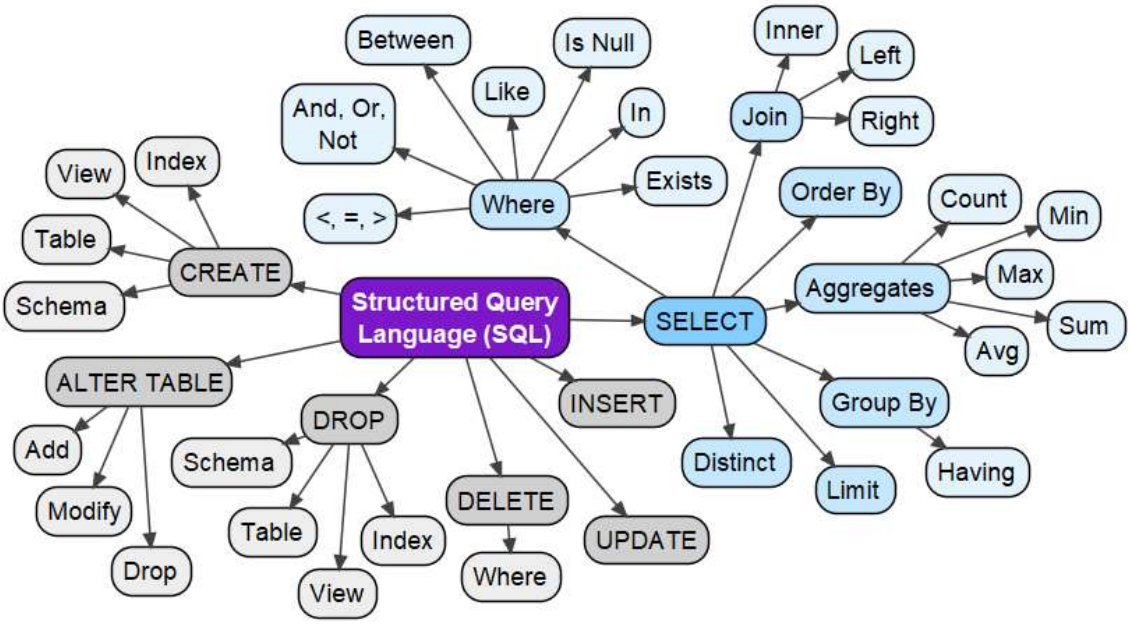


Figure 82. Overview of SQL SELECT Statements

In Figure 82, we see an overview of all the variations on `SELECT` statements that we are going to learn about in this learning unit.

2 Basic Queries

2.1 SELECT Statement

Definition

"The **`SELECT`** statement, or command, allows the user to extract data from tables, based on specific criteria." (Watt & Eng, 2014)

We already saw the most basic `SELECT` statement in Learning Unit 3 when we were inserting data into the `racetrack` table (see Figure 83).

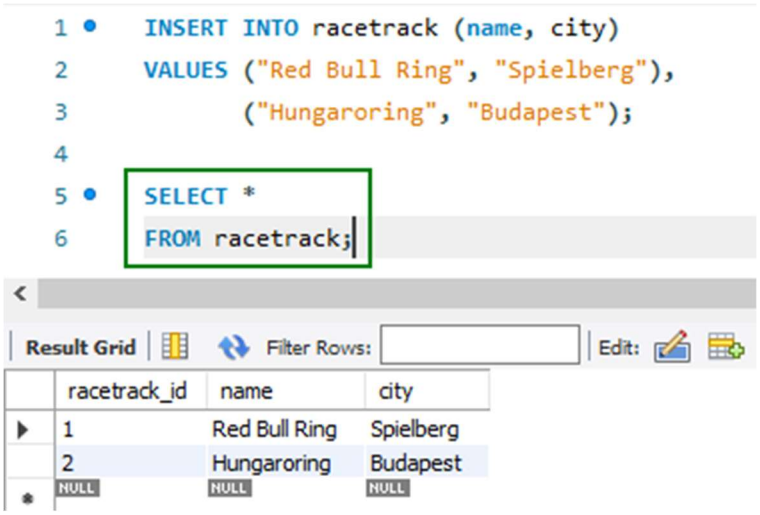


Figure 83. INSERT INTO and SELECT

The `SELECT` keyword indicates that we are querying data, and it is followed by the columns that we want to get. In this most basic query, we specified `*`. That means getting all the columns from the table. The `FROM` keyword specifies where the data is originating from – the table `racetrack` in this case.

In short, this query gets all the rows and all the columns from the `racetrack` table.

2.1.1 LIMIT

When this query is executed using MySQL Workbench, you might notice something interesting if you look closely at the output window. The query that is executed has an additional statement added to it: `LIMIT 0, 1000` (see Figure 84).

Definition

"**`LIMIT`** is a special clause used to limit MySQL records a particular query can return." (BitDegree, 2019)

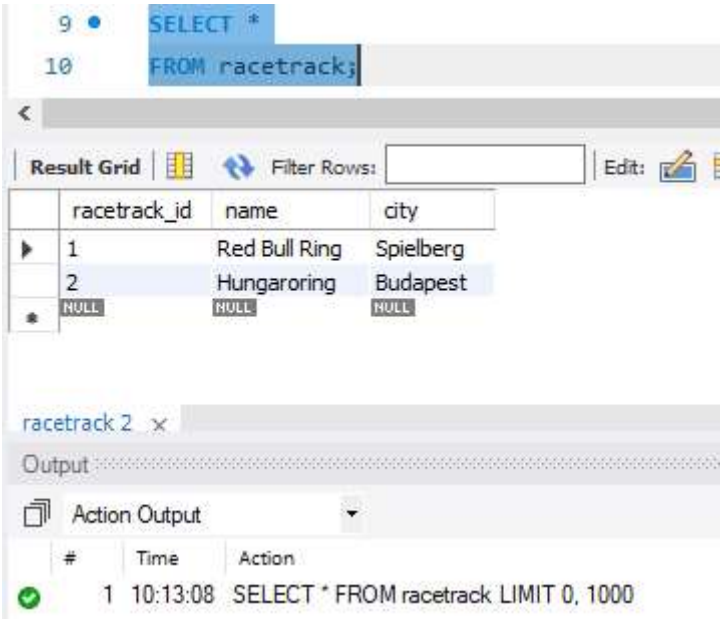


Figure 84. SELECT with LIMIT

The first parameter (0 here) is the row to start at. The second parameter (1000) is how many rows to include.

MySQL Workbench adds this clause to avoid getting back and having to handle huge data sets. We only have two racetracks on our table right now. But imagine how much resources it would take if you had a million clients and requested all that data.

We could also add this same statement to our own queries. For example, if you wanted to display only five racetracks on a webpage, you could limit the number of records that you query to five. And then for the next page, start at a row offset of five.

2.1.2 Specifying Column Names

Depending on what the retrieved data is going to be used for, you might decide that you don't need all the columns. Say for example we wanted to get only the name and date of all races in the database. Instead of specifying *, specify a comma separated list of column names.


```
1 •   SELECT name, date
2     FROM race;
```

Figure 85. SELECT with column names

The columns will appear in the result set in the order that they are specified in the query. This does not change the meaning of the data, but it might mean that the data is displayed in a more user-friendly order.

What if we wanted the columns to be called `racename` and `racedate` instead? When working with column names (and even table names), an alias can be specified.

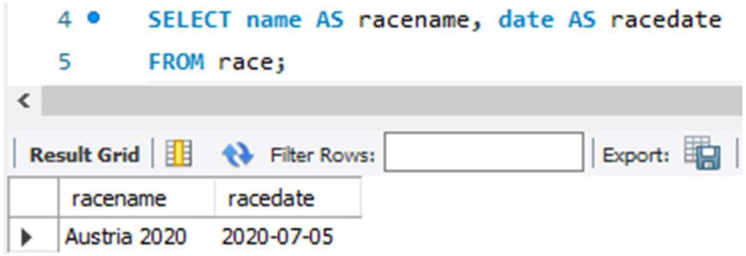


Figure 86. SELECT with column aliases

We see the new column names here in MySQL Workbench. But that is not the only place where it is useful. If you were to call this query from your Java or C# code, you would still reference these new column names.

We can also give a table an alias. This is usually used to shorten the name of the table, to make queries more concise. Figure 87 (line 8) shows what this looks like.

```
7 •   SELECT r.name AS racename, r.date AS racedate
8     FROM race r;
```

Figure 87. SELECT with table alias

Right now, with a single table, that is not so useful yet. But you will see later when we discuss joins why this is important.

2.2 WHERE Clause

So far, we have seen how to get only certain columns as a part of the result set. But what if you wanted to specify criteria that rows must match to be included? That is the purpose of the `WHERE` clause.

There are lots of ways to filter rows, as shown in Figure 88. Let's start with the most basic operators: <, = and >.

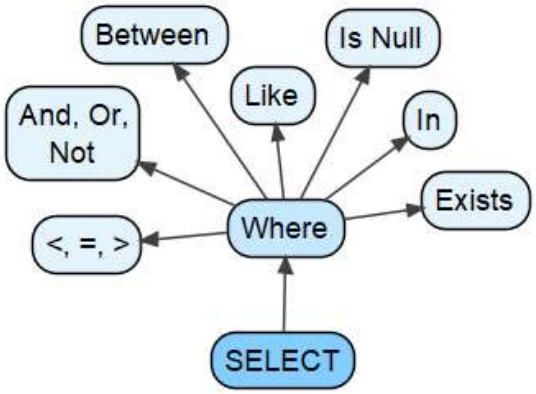


Figure 88. WHERE clause possibilities

2.2.1 Less Than, Equals, Greater Than Operators

Remember when we were updating a specific row's data in Learning Unit 3 (see Figure 89)? We made use of a `WHERE` clause there too, with the equals operator. What happened in the background is that the rows in the table were filtered according to what the WHERE clause specified (in that case, for a specific `race_id`). And only after that the `UPDATE` got executed.

```
1 •   UPDATE race
2     SET date = "2020-07-12"
3     WHERE race_id = 2;
```

Figure 89. UPDATE with WHERE clause

The same thing happens when you have a `SELECT` query with a `WHERE` clause. The rows get filtered and, in this case, then the results are returned to the user.

> Notice that the equals operator here is a single equals sign, unlike Java and C# where it would be == for a comparison.

Let us `INSERT` the Hungary 2020 race again – the one that we deleted before. And a couple of 2019's races too, so we have more data to work with.

```
12 •   SELECT *
13     FROM race;
```

Result Grid | Filter Rows: | Edit:

race_id	racetrack_id	name	date
1	1	Austria 2020	2020-07-05
3	2	Hungary 2020	2020-07-19
4	2	Hungary 2019	2019-08-04
5	1	Austria 2019	2019-06-30

Figure 90. More Race Data

If we wanted to find the `Race` with `race_id` 3, we would get back only a single record. This is because the column is a primary key, which by its very nature is unique.

```
1 •  SELECT *
2    FROM race
3    WHERE race_id = 3;
```

Result Grid | Filter Rows: | Edit

race_id	racetrack_id	name	date
3	2	Hungary 2020	2020-07-19

Figure 91. Race with race_id of 3

But say we wanted to find all the races that took place at the `racetrack` that has a `racetrack_id` of 2. Foreign keys are not necessarily unique, and in fact in most cases, are not. Then we would get two records from our current data set.

```
5 •  SELECT *
6    FROM race
7    WHERE racetrack_id = 2;
```

Result Grid | Filter Rows: | Edit

race_id	racetrack_id	name	date
3	2	Hungary 2020	2020-07-19
4	2	Hungary 2019	2019-08-04

Figure 92. Races held at racetrack 2

Equals is not the only operator we can use, though – there is also >, <, >= and <=. These work as you would expect in any programming language. For example, if we wanted all the races that took place since 1 January 2020, we could write the query shown in Figure 93.

```
13 •  SELECT *
14    FROM race
15    WHERE date >= "2020-01-01";
```

Result Grid | Filter Rows: | Edit

race_id	racetrack_id	name	date
1	1	Austria 2020	2020-07-05
3	2	Hungary 2020	2020-07-19

Figure 93. Races since 1 January 2020

2.2.2 LIKE Operator

The LIKE operator is used to write queries where values are compared to wildcards. For example, if we wanted all the races with names that start with an A, we could write the following query:

```
26 •  SELECT *
27    FROM race
28    WHERE name LIKE "A%";
```

Result Grid | Filter Rows: | Ed

race_id	racetrack_id	name	date
1	1	Austria 2020	2020-07-05
5	1	Austria 2019	2019-06-30

Figure 94. Query with LIKE operator

Here we see the wildcard %, which can match any number of letters. So, the name must have an A at the start, and then have any number of letters after that.

There is also a second wildcard: _. The underscore matches only one character. (mysqltutorial.org, 2020)

2.2.3 IS NULL Operator

So far, we have only had mandatory properties. But let us look at the `CREATE TABLE` script for the `Entry` table.

```
18 • CREATE TABLE entry (
19       entry_id     INT AUTO_INCREMENT NOT NULL,
20       team_id      INT NOT NULL,
21       race_id      INT NOT NULL,
22       driver_id    INT NOT NULL,
23       position     INT,
24       PRIMARY KEY (entry_id),
25       FOREIGN KEY (team_id) REFERENCES team(team_id),
26       FOREIGN KEY (race_id) REFERENCES race(race_id),
27       FOREIGN KEY (driver_id) REFERENCES driver(driver_id)
28   );
```

Figure 95. Creating the table entry

Here we see that the position can be null. Why? Well, drivers enter a race long before it happens. And we want to be able to create the entry when it is made. Then, when the race is over, we can update the values according to where the drivers are placed.

How can we find all the rows where the position has not been added yet? By using the `IS NULL` operator.

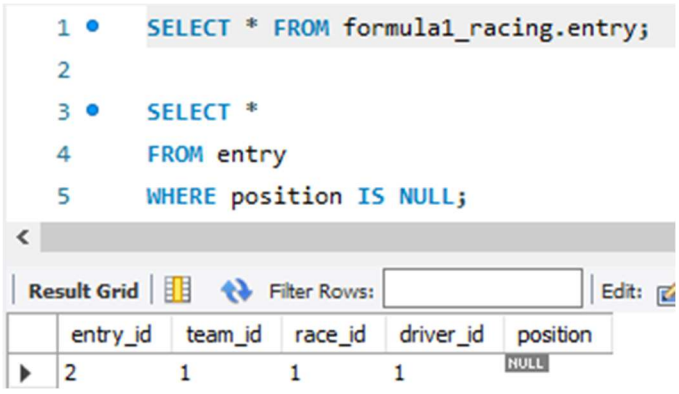


Figure 96. Getting the entries where position is null

2.2.4 IN Operator

The `IN` operator allows you to compare the value for a specific attribute to a set of values. (Oracle Corporation, n.d. g)

For example, if we want to find the list of races with their IDs in the set (3, 4), the query would look as shown in Figure 97.

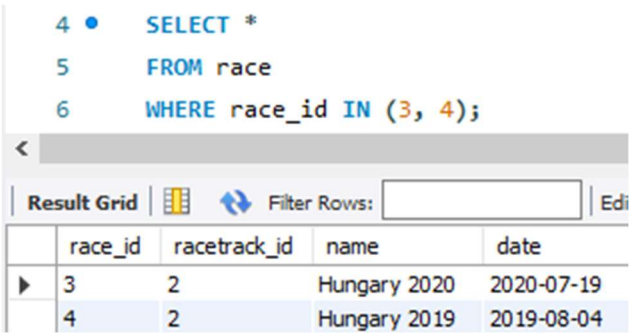


Figure 97. Finding races 3 and 4

2.2.5 EXISTS Operator

The `EXISTS` operator works with a subquery, and the condition will become true if the subquery returns at least one row. (TechOnTheNet.com, n.d.)

Say we wanted to find all the teams that have made at least one entry. We could write a subquery that gets all the entries for the team, and if it returns some rows then we include the team in the result set. This is shown in Figure 98.

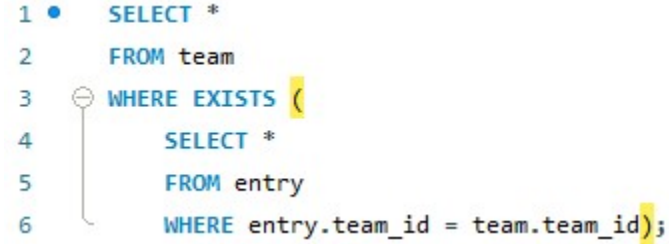


Figure 98. Finding all teams with at least one entry

This is not very efficient at all though since that subquery gets run for every row in the team table. (TechOnTheNet.com, n.d.) We can write more efficient queries with joins – see Section 3.

2.2.6 Logical Operators

Just like in Java or C# programming, we can also construct more complicated comparisons by using the logical operators `AND`, `OR` and `NOT`.

> **Note:** In SQL, we write out these operators in words. It is literally `AND`, `OR` and `NOT` in our queries.

If we wanted to find only the races that took place in June 2019, we could say that it is all the races where the date is greater than or equal to 1 June 2019, `AND` less than or equal to 30 June 2019. That means that the interval includes the start and end dates. That would look like the query shown in Figure 99.

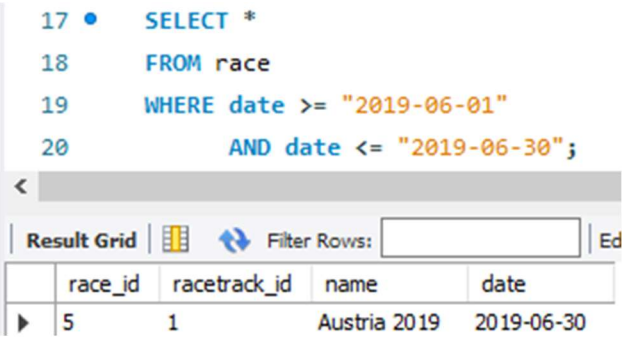


Figure 99. Races in June 2019

2.2.7 BETWEEN Operator

The case that we looked at with the start and end date of an interval that is specified, is a case that arises so often in SQL databases that there is a special keyword for it: `BETWEEN`. When using this keyword, the lower and upper bounds are included. This means that it can be used in place of a combination of greater than or equals and less than or equals operators. (tutorialspoint.com, 2020)

We can then rewrite the June 2019 query as shown in Figure 100.

```
22 •  SELECT *
23    FROM race
24    WHERE date BETWEEN "2019-06-01" AND "2019-06-30";
```

Figure 100. Query with BETWEEN

The advantages of writing a query with `BETWEEN` are that it is easier to understand and easier to get right. With the `>= AND <=` syntax, it is entirely possible to have the operators the wrong way around. That is much less likely with `BETWEEN`.

2.3 SELECT DISTINCT Statement

Values can repeat in our data; for example, multiple teams may come from the same country. To get the list of countries where the teams come from **without duplicates**, we could use the `SELECT DISTINCT` statement.

```
3 •   SELECT DISTINCT(country)
4     FROM team;
```

Figure 101. Finding the countries where teams come from

Note that having to use the `SELECT DISTINCT` statement often means that the design is less than optimal. In this case, we could create a `country` table and store just the FK in the team for example. Then the `country` table would already have the list of distinct countries.

2.4 Ordering of Results

If you run a `SELECT` query, the results will be returned in the order that the rows are stored in the database. And that can be fine in some cases. But very often, we want to order the results in a different way. For example, we might want to get the teams ordered alphabetically by name.

```
1 •   SELECT *
2     FROM team
3     ORDER BY name ASC;
```

Figure 102. Teams ordered by name

Here we see the `ORDER BY` clause with a single column. And the values are being sorted in ascending order (smallest to largest). This is specified using `ASC` here but is also the default that will be used if the ordering is not specified. If you want to sort the data in descending order instead, use `DESC`.

```
5 •   SELECT *
6     FROM team
7     ORDER BY country, city;
```

Figure 103. Order by country then city

Data can also be sorted according to multiple columns. For example, here the rows are first sorted by country, and then by city.

3 Joining Tables

3.1 Joining Tables

We have now looked at several different ways to filter and sort the data from a single table. But that is not super exciting. Most of the tables in our design are related to other tables. And it must be possible to get the data from the related tables too. To do that, we must join tables.

Let us look at an example: getting the names of all the races, together with the name of the racetrack where each one took place. The name of the race is in the `race` table, and the name of the racetrack is in the `racetrack` table. So, we must combine data from `race` and `racetrack` to get the answer we are looking for.

There are two ways in which this can be expressed in SQL. The first way is to just use a `WHERE CLAUSE`.

```
1 •   SELECT race.name as racename, racetrack.name as racetrackname
2     FROM racetrack, race
3     WHERE race.racetrack_id = racetrack.racetrack_id
4     ORDER BY racename;
```

Figure 104. Join with a where clause

Here we see that we can specify that there are two tables that are the sources of our data. And we are only interested in those rows where the `racetrack_id` from both tables match up. That was the very purpose of the FK that we created in `race` after all.

We mentioned before that we can have an alias for each table in a query. Let's see how that shortens this query.

```
6 •   SELECT r.name as racename, t.name as racetrackname
7     FROM racetrack t, race r
8     WHERE t.racetrack_id = r.racetrack_id
9     ORDER BY racename;
```

Figure 105. Join with table aliases

This is one way of writing a join, and it is also an older way of doing things. The newer way of writing this join uses the JOIN keyword.

```
11 •  SELECT r.name as racename, t.name as racetrackname
12    FROM racetrack t
13    JOIN race r ON t.racetrack_id = r.racetrack_id
14    ORDER BY racename;
```

Figure 106. Using the JOIN keyword

These two queries return the same result set. But the benefit of using the `JOIN` keyword is what you can clearly see that the purpose of the keys being equal is to do the joining of the tables. If we now added a `WHERE` clause that says filters by date, the `JOIN` and the `WHERE` clearly have different purposes. And this makes the query easier to read.

```
17 •    SELECT r.name as racename, t.name as racetrackname
18      FROM racetrack t
19      JOIN race r ON t.racetrack_id = r.racetrack_id
20      WHERE r.date >= "2020-01-01"
21      ORDER BY racename;
```

Figure 107. Join with a WHERE clause to filter

3.2 Types of Joins

3.2.1 Inner Joins

The joins that we have done so far are **inner joins** – one of the most frequently used types of join. To understand the different types of joins, it is useful to think of the two tables where our data is coming from, as sets. As shown in Figure 108, an inner join is where matching rows from both tables exist. So, in set theory that would be the intersection of the two sets.

In our races and racetracks example above, we will only get data back where there is a race that has taken place at a specific racetrack. And neither of the names will hence be null.

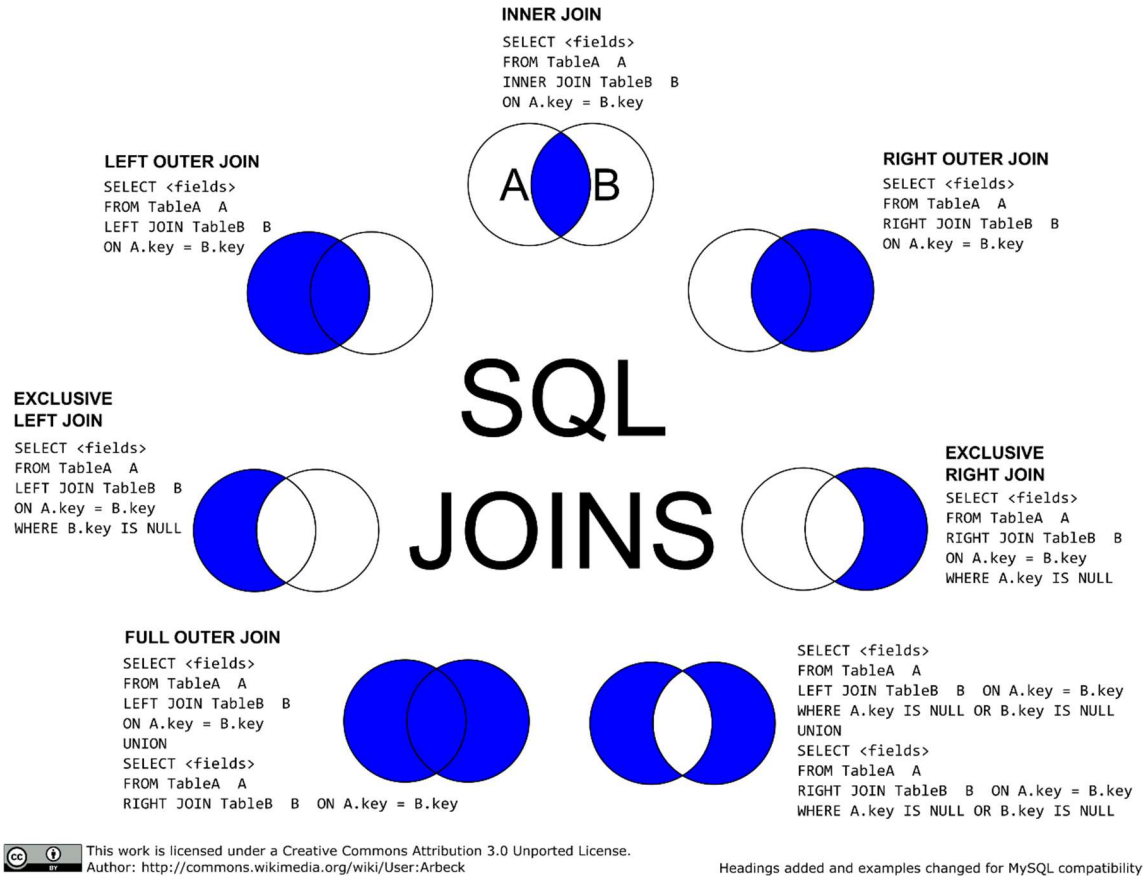


Figure 108. Types of Joins (based on (Beck, 2013))

3.2.2 Left and Right Outer Joins

A `LEFT` join includes all the rows from the left table (the first one that is specified), and any rows that match from the right table. In our example, `racetrack` was the left table and `race` was the right table. So, what if we wanted to also include in the results any racetracks that don't have races recorded against them yet?

```
1 •  SELECT t.name as racetrackname, r.name as racename
2    FROM racetrack t
3    LEFT JOIN race r ON t.racetrack_id = r.racetrack_id
4    ORDER BY racename;
5
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content

racetrackname	racename
Circuit de Monaco	NULL
Red Bull Ring	Austria 2019
Red Bull Ring	Austria 2020
Hungaroring	Hungary 2019
Hungaroring	Hungary 2020

Figure 109. LEFT join

Now we have Circuit de Monaco included, with NULL for the `racename`. That is typical of a `LEFT` join – where there is not a matching record, the values from the other table will be NULL.

The RIGHT join works in the same way, except then all the records from the right table gets included, together with anything from the left table that matches.

3.2.3 Exclusive Left and Right Joins

Looking at Figure 109, we see that an exclusive left join is where a row occurs in the left table and not in the right table. If we wanted to find **only** the racetracks that do not have races yet, this is the way to do it.

```
1 •  SELECT t.*
2    FROM racetrack t
3    LEFT JOIN race r ON t.racetrack_id = r.racetrack_id
4    WHERE r.racetrack_id IS NULL;
5
```

Result Grid | Filter Rows: | Export: | Wrap Cell Cont

racetrack_id	name	city
3	Circuit de Monaco	Monte Carlo

Figure 110. Racetracks with no races

3.2.4 Full Outer Join

A full outer join is where all the records from both tables are included. MySQL does not support the `FULL OUTER JOIN` keyword, but it can be accomplished by taking the `UNION` of the `LEFT` and `RIGHT` joins.

```
1 •  SELECT *
2    FROM racetrack t
3    LEFT JOIN race r ON t.racetrack_id = r.racetrack_id
4
5    UNION
6
7    SELECT *
8    FROM racetrack t
9    RIGHT JOIN race r ON t.racetrack_id = r.racetrack_id;
```

Figure 111. Full outer join using `UNION`

4 Aggregate Functions

4.1 What is an Aggregate Function?

All the queries that that we have written so far returned result sets that contained rows (or parts of rows) of the data from our tables. And these queries are super useful. But … what if we wanted to calculate values across different rows in our tables? Say, for example, we wanted to find out how many races are recorded in the database. That is a **calculation** that works based on **lots of rows**, not just one. And that is where the aggregate functions come in.

Figure 112 shows a list of some of the most common aggregate functions that are available in MySQL.

`AVG()`	Calculates and returns the average of the values.
`COUNT()`	Calculates and returns the number of rows that are included in the input values set.
`MAX()`	Finds and returns the highest value in the input values.
`MIN()`	Finds and returns the lowest value in the input values.
`STDDEV()`	Calculates and returns that standard deviation of the input values.
`SUM()`	Calculates and returns the total of the elements in the set of input values.

Figure 112. MySQL Aggregate Functions
(Oracle Corporation, n.d. h)

Note that the exact names of these functions do vary according to the dialect of SQL that you use. So, the above list is going to be useful to us when working with queries in MySQL.

4.2 Using Aggregate Functions

Aggregate functions can be used on their own (without a `GROUP BY` clause). If we do that, all the records are considered part of one big group. We will talk more about `GROUP BY` in the next section, and then that statement will become clearer. But for now, let's look at using an aggregate function on a whole table.

Let us say we want to find out how many races are stored in our `race` table.

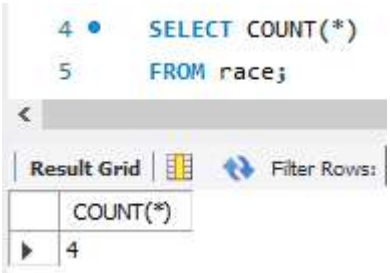


Figure 113. Counting all the races

So far, our race table only has four races …

This is a great example of where you might decide to use an alias for a column. Although `COUNT(*)` is meaningful to use as SQL developers, the end users would appreciate a more meaningful name. For that reason, it is useful to add an easy-to-understand alias to the calculated column.

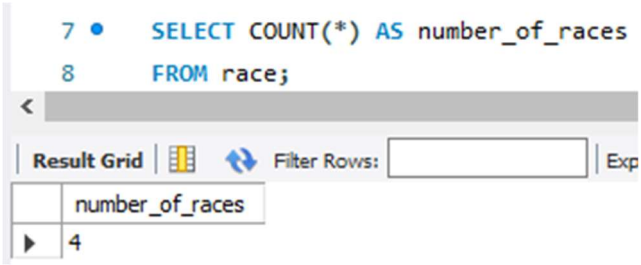


Figure 114. Count with an alias

Here we used `COUNT(*)` – it returns the number of rows, independent of whether there are any null values or not.

We could also use `COUNT()` on a specific column. For example, if we wanted to find out how many `positions` have been stored in the `entry` table (remember that is an optional field, so it can be null), we could use `position` as a parameter as shown in Figure 115.

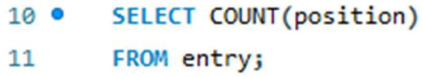


Figure 115. Counting the number of positions stored in the entry table

This query counts the rows with non-null `position` values.
All the other aggregate functions need a column. For example, calculating the minimum position that a driver has attained would require passing the position column to the `MIN()` function.

It is useful to note that the rows can also be filtered first before performing an aggregate function. For example, we could add a `WHERE` clause that filters by date. Now the `COUNT(*)` function returns 2, because the rows were filtered first before the aggregate function was applied.

```
16 •    SELECT COUNT(*)
17      FROM race
18      WHERE date >= "2020-01-01";
```

Figure 116. Counting the races from 1 January 2020

5 Grouping Records

5.1 GROUP BY

In the discussion on aggregate functions, we mentioned the fact that the whole table is considered one big group if you do not use `GROUP BY`. So, what is `GROUP BY`? It is a way of splitting the records into a group.

What if we wanted to count how many races have been held at each racetrack? Well, if we wanted to do that manually, we would group the races according to which racetrack each was held at. And then count each of those sets of races to figure out how many there have been.

The SQL syntax to do this does a similar thing. We get all the races and racetracks, group those according to the track name, and then count the number of rows in each group.

```
1 •    SELECT t.name, COUNT(*)
2      FROM race r
3      JOIN racetrack t ON r.racetrack_id = t.racetrack_id
4      GROUP BY t.name;
5
```

Result Grid | Filter Rows: | Export: | Wrap Cell Co

name	COUNT(*)
Red Bull Ring	2
Hungaroring	2

Figure 117. Number of races held at each racetrack

Tip: When writing a query like this, start simple. First, just write a normal `SELECT` statement that gets all the races and racetracks. Then add the `GROUP BY` and aggregate functions.

If we have a `GROUP BY`, we can still filter the rows first by adding a `WHERE` clause.

```
6 •  SELECT t.name, COUNT(*)
7    FROM race r
8    JOIN racetrack t ON r.racetrack_id = t.racetrack_id
9    WHERE r.date >= "2020-01-01"
10   GROUP BY t.name;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Co

name	COUNT(*)
Red Bull Ring	1
Hungaroring	1

Figure 118. Query with WHERE and GROUP BY

5.2 HAVING

We saw in the previous example that we can filter the rows first before grouping using a `WHERE` clause. But what if we wanted to filter the results based on the aggregate function values? We can make use of the `HAVING` clause for that.

`HAVING` works like a `WHERE` clause in terms of the conditions that it can specify. Say, for example, we want to get the number of races held at each racetrack, but only get back the data for racetracks that have more than 10 races. We can do that using a `HAVING` clause.

```
1 •  SELECT t.name, COUNT(*) AS number_of_races
2    FROM race r
3    JOIN racetrack t ON r.racetrack_id = t.racetrack_id
4    WHERE r.date >= "2020-01-01"
5    GROUP BY t.name
6    HAVING number_of_races > 10;
```

Figure 119. HAVING clause

We have looked at lots of different ways to query data, and many of those can be used in conjunction with one another. But there is an order in which all these operations happen when they are combined in the same query. We have seen some of it already when we said that the `WHERE` clause gets executed before we get to the `GROUP BY`. But there is more to it than that. The order of execution is shown in Figure 120.

Figure 120. Order of execution in queries (based on (Shay, 2018) and (SQLBolt, 2019))

6 Recommended Additional Reading

MySQL Tutorial is a great website with a lot of resources about using MySQL. It is available here: https://www.mysqltutorial.org/ [Accessed 11 December 2023].

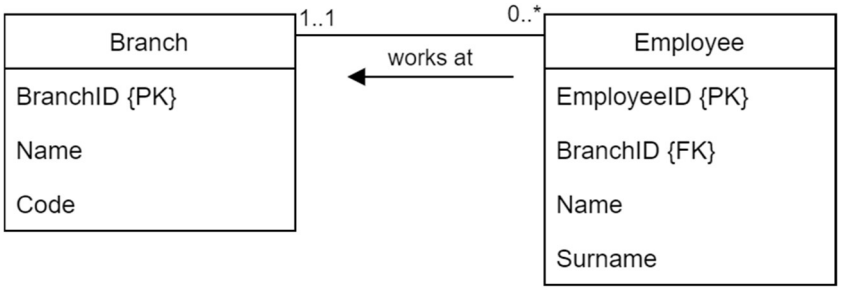
7 Revision Exercises

7.1 Revision Exercise 1

What is the difference between `WHERE` and `HAVING`?

7.2 Revision Exercise 2

Assume the below ERD has been implemented in a SQL database. Write a query that lists all the employees with their name, surname, and the name of the branch where they work.



8 Solutions to Revision Exercises

8.1 Revision Exercise 1

Read Section 5.2 above.

8.2 Revision Exercise 2

Read Section 3 above.

Learning Unit 5: Indexes, Views, Temporary Tables and Stored Procedures	
Learning Objectives: • Explain the purpose of an index. • Create an index using SQL. • Explain the purpose of a view. • Create a view using SQL. • Explain the purpose of a stored procedure. • Create a stored procedure using SQL. • Explain the purpose of a temporary table. • Identify the types of temporary tables. • Use temporary tables in queries.	**My notes**
Material used for this learning unit: • This module manual. • MySQL server installation.	
How to prepare for this learning unit: • Read through the content in this learning unit.	

1 Introduction

In the previous two learning units, we covered all the basics that you need to get started with creating and using a SQL database. Looking at Figure 121, we have done most of what we set out to do.

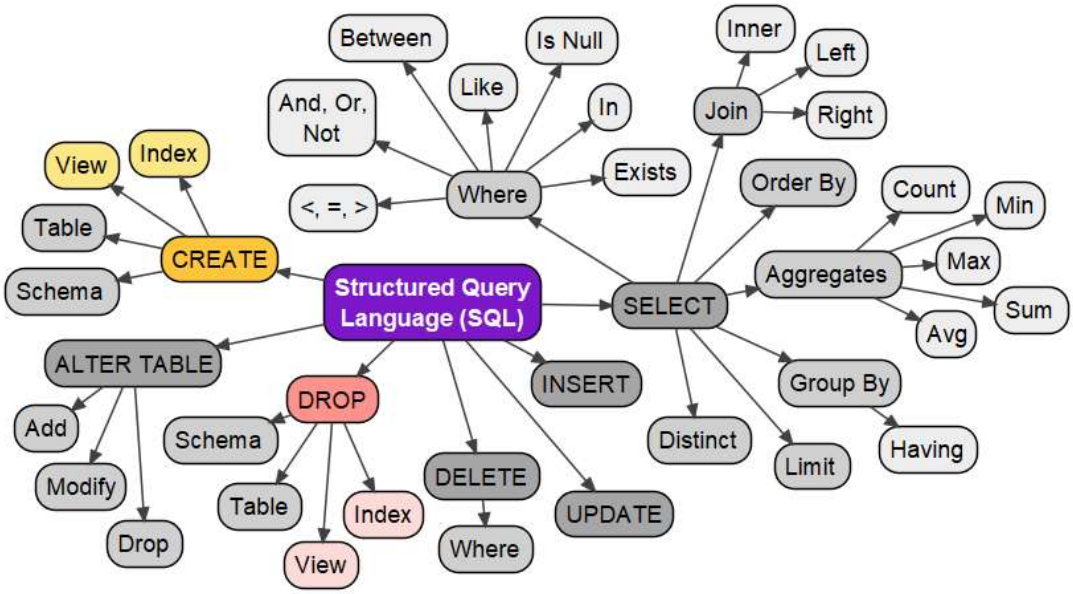


Figure 121. Overview of SQL Statements

This learning unit will deal with some more advanced topics: indexes, views, stored procedures, and temporary tables.

2 Indexes

2.1 What is an Index?

Definition
"An **index** is an on-disk structure associated with a table or view that speeds retrieval of rows from the table or view. An index contains keys built from one or more columns in the table or view." (Guyer, et al., 2019)

The data that we have worked with so far is tiny – only a couple of races, drivers, and racetracks. But if your database contains hundreds of thousands of products, or millions of customer transactions, then performance starts to become a concern when writing queries.

That is where indexes come in. For example, for an online retailer, customers will often search by product category. So, we want to be able to find products by category as fast as possible.

Indexes are maintained for all primary keys, so searching by a primary key is already a fast operation. And additionally, there are `UNIQUE`, `INDEX` and `FULLTEXT` indexes. All of these are stored in a **B-tree** data structure in MySQL. (Oracle Corporation, n.d. i)

The details of a B-tree are beyond the scope of this module but suffice to say that it is "a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time." (Tang, 2012) So, the point of the B-tree data structure is once again to make searches as fast as possible.

So, what operations are improved by using indexes? MySQL uses indexes, if available, for the following operations (Oracle Corporation, n.d. i):

- Filtering rows according to a `WHERE` clause;
- Joining tables;
- Finding the `MIN()` or `MAX()` value in a group; and
- Sorting and grouping.

In short, if we have a large table and only a few of the rows typically need to be returned by a query, then having an index will improve the performance of the query. But if the table is small, just reading all the rows sequentially might even be slightly more efficient. (Oracle Corporation, n.d. i)

2.2 Types of Indexes

There are two types of indexes in SQL databases: **clustered** and **non-clustered**.

A **clustered index** is one where the table data is physically stored in the order that the index determines. For the primary key of a table, a clustered index is created by default. And because this kind of index determines the order in which the data is stored, each table can only have **one** clustered index. (Tang, 2012)

The other type of index is a **non-clustered index**. Tables can have **multiple** non-clustered indexes since the ordering is stored in a separate data structure. (Tang, 2012) This means that using a non-clustered index is a trade-off between faster data access and using up more storage. (Guru99, 2020c)

Read (Guru99, 2020c) for a more detailed comparison of clustered and non-clustered indexes.

2.3 Creating an Index

An index can be created on a single column, or multiple columns. (Bradley, 2018)

When creating a table using the `CREATE TABLE` statement, all the indexes can already be defined. Let us look again at the very first `CREATE TABLE` statement that we discussed in Learning Unit 2.

```
1  CREATE TABLE racetrack (
2      racetrack_id    INT AUTO_INCREMENT NOT NULL,
3      name            VARCHAR(250) CHARACTER SET utf8mb4 NOT NULL,
4      city            VARCHAR(50) CHARACTER SET utf8mb4 NOT NULL,
5      PRIMARY KEY (racetrack_id)
6  );
```

Figure 122. CREATE TABLE for a single table

Here we specified the primary key of the table – the surrogate key called `racetrack_id`. Remember, in the background, a clustered index automatically gets created for that PK. Let us look at the details of the table as displayed by MySQL Workbench to see what the effect of defining `racetrack_id` as the PK was.

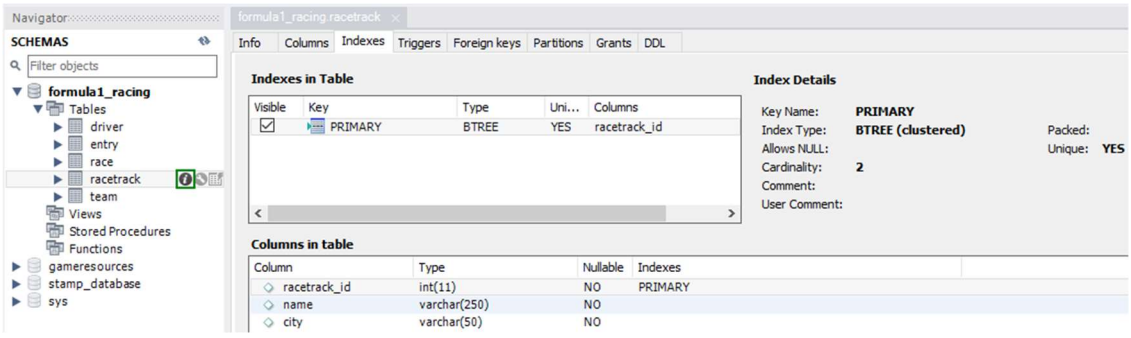


Figure 123. Indexes of the Racetrack Table

We see a whole lot of interesting things in Figure 123. Firstly, we see that the index did indeed get created automatically, like the theory said it should. We also see that it is a clustered index, which means that the data is stored in the order imposed by the PK. And that it is in fact stored in a B-tree.

What if we discover after the table is already created and has a lot of data that we need another non-clustered index? This is where the `CREATE INDEX` statement is useful.

```
1 •   CREATE INDEX racetrack_city
2     ON racetrack (city);
```

Figure 124. Creating a new index

The `CREATE INDEX` statement in Figure 124 creates an index on the `city` column of the `racetrack` table. And here we do have the optional name `racetrack_city` which is useful to identify a specific index.

There is also a `DROP INDEX` statement for removing an index.

3 Views

3.1 What is a View?

Definition
A **view** is “a virtual table whose contents (columns and rows) are defined by a query.” (Milener, et al., 2020)

Another interesting way of describing a view is that it is like a window that allows you to look at a part of the data. (Soderberg, 2019)

The fact that a view is considered a virtual table, means that a view can be used anywhere in queries where a normal table would be used. But why have such a virtual table? A view is useful when (Milener, et al., 2020):

- A specific user needs to have a certain view on the data;
- Users should not have access to the tables due to security considerations; and

- A table changes and backward compatibility with existing software needs to be maintained.

How does the data for a view get stored? For a non-indexed view, it does not get stored in a separate place at all – the query gets executed when it is used. That means that the data is dynamic and always up to date. (Guyer, et al., 2017)

3.2 Creating a View

We know that a view is based on a query. So, write the query first and execute it until you are 100% happy with it. And then just add the `CREATE VIEW` statement at the top.

Figure 125 shows an example of creating a view that will either create the view if it does not exist or replace it if it does. Replacing a view is a very common use case since the requirements of what users want to see often change.

Replacing the view only changes the query. No data changes are done at all, and the data will just get dynamically queried with the new query for the next time that it is used.

```
1 •  CREATE OR REPLACE VIEW races_with_racetracks AS
2    SELECT r.*, t.name as racetrack
3    FROM race r
4    JOIN racetrack t ON r.racetrack_id = t.racetrack_id;
```

Figure 125. Creating a view

How do we use the view? Just like we would a table. For example, we could write the query shown in Figure 126 that gets all the rows from the view and then apply a `WHERE` clause to the results.

```
6 •  SELECT *
7    FROM races_with_racetracks
8    WHERE race_id > 2;
```

Result Grid | Filter Rows: | Export: | Wrap C

race_id	racetrack_id	name	date	racetrack
3	2	Hungary 2020	2020-07-19	Hungaroring
4	2	Hungary 2019	2019-08-04	Hungaroring
5	1	Austria 2019	2019-06-30	Red Bull Ring

Figure 126. Using the view in a query

Tip: Joining a view can be done. But do be aware that, depending on what is in the query of that view, the resulting join may be very inefficient indeed. So, it is not recommended to just join on a view without considering the details of the specific use case.

4 Stored Procedures

4.1 What is a Stored Procedure?

If we have a `SELECT` query that we use a lot, we can use a view to give that query a name. And that means we can easily call it as we have already seen above. But what if we have other SQL statements that get called very often? How can we make those reusable? By creating a **stored procedure**. (Soderberg, 2019)

There are several **benefits** to using stored procedures (mysqltutorial.org, 2020b):

- There is **less traffic** between applications and the database, because now one call to a stored procedure replaces multiple other calls to the database;
- The **business logic** is contained in one **central place**: in the database itself; and
- **Security** can be better since an application would only need access to the stored procedures that it uses.

However, there are also drawbacks to this approach (mysqltutorial.org, 2020b):

- Lots of stored procedures means that each **database connection** uses lots of **memory**;
- Issues in a stored procedure are very **hard to debug**; and
- **Maintaining** stored procedures requires **special skills** that not everybody has.

4.2 Creating a Stored Procedure

Creating a stored procedure involves using the `CREATE PROCEDURE` statement. For example, in our database adding a race is something that will happen quite often. To be able to insert the race though, we first need to find out what the racetrack's id is if we only have the name of the racetrack, so that we can insert that as the FK. Figure 127 shows how to create this stored procedure.


```
1     -- Change the delimiter to be able to include ; in
2     -- the statements in the PROCEDURE body
3     DELIMITER //
4
5     -- Declare the name and parameters of the PROCEDURE
6     CREATE PROCEDURE AddRace
7     (IN racename VARCHAR(250),
8      IN racedate DATE,
9      IN racetrackname VARCHAR(250))
10    BEGIN
11
12        -- Declare a local variable to store the racetrack's id
13        DECLARE racetrack_id_to_insert INT;
14
15        -- Get the id of the racetrack
16        SELECT racetrack_id
17        INTO racetrack_id_to_insert
18        FROM racetrack
19        WHERE name = racetrackname;
20
21        -- Insert the new race
22        INSERT INTO race(racetrack_id, name, date)
23        VALUES (racetrack_id_to_insert, racename, racedate);
24
25    END //
26
27    -- Change the delimiter back
28    DELIMITER ;
```

Figure 127. Creating a stored procedure

The important gotcha is that we need to change the delimiter – the character that is used to end a SQL statement before we start to declare the stored procedure. (Oracle Corporation, n.d. j) This is useful since we will have multiple statements that should just be stored as part of the body of the stored procedure. And then we need to remember to change it back to; again at the end of the script.

Creating a stored procedure works like declaring a method in C# or Java. The procedure has a name that we will use to call it, and it can have parameters. In this case, we have three input parameters: `racename`, `racedate` and `racetrackname`. Note that the type of each of the parameters is specified after the name, unlike Java or C#.

The `BEGIN` and `END` keywords serve the same purpose as the curly braces `{}` in C# and Java. This indicates where the procedure starts and ends.

The first order of business is to find out what the ID of the racetrack is. That we do, as you might expect, but using a `SELECT` statement. The new thing here, though, is that we select that ID value `INTO` a variable that we declare in the stored procedure. This, of course, assumes that the racetrack already exists.

Now, we have all the information, we need to call the `INSERT INTO` statement to create the new race.

4.3 Calling a Stored Procedure

Just like with a method, the stored procedure will not accomplish anything unless we call it. How do we do that? By making use of the `CALL` statement.

```
32 • CALL AddRace(
33       "New race via procedure",
34       "2020-08-02",
35       "Circuit de Monaco");
```

Figure 128. Calling the Stored Procedure

The three arguments are passed to the stored procedure just like the arguments would be passed to a method in Java or C#. And then all the statements in the body of the stored procedure gets executed.

There is a lot more that can be said about stored procedures. Read (mysqltutorial.com, 2020c) for more details.

5 Temporary Tables

5.1 What is a Temporary Table?

A temporary table is a table that is, well, temporary. It is used to store data for a short period, after which it is deleted again. Temporary tables will automatically get dropped when the user disconnects. (JavaTpoint, n.d.)

Unlike normal views, in a temporary table the values do get stored.

5.2 Types of Temporary Tables

There are two ways in which temporary tables can be stored: in memory or on disk. (Olamendy, 2017) In-memory temporary tables are faster for small data sets but use memory, which is far more limited than disk space. So, it is a trade-off between speed and memory usage.

Using a temporary table that is stored on disk is still fast, especially since these tables do support indexing. So, for larger data sets, using a temporary storage on disk works well.

5.3 Using Temporary Tables

A temporary table can be created **explicitly**, using the same syntax as a normal table just, with the `TEMPORARY` keyword added. The one difference is that we cannot specify foreign key constraints here.

```
1  CREATE TEMPORARY TABLE current_season_races (
2      race_id         INT AUTO_INCREMENT NOT NULL,
3      racetrack_id    INT NOT NULL,
4      name            VARCHAR(250) CHARACTER SET utf8mb4 NOT NULL,
5      date            DATE NOT NULL,
6      PRIMARY KEY (race_id)
7  ) ENGINE=MEMORY;
```

Figure 129. Explicitly Creating In-Memory Temporary Table

In the `CREATE TEMPORARY TABLE` statement in Figure 129, we see that the `ENGINE` can be specified. In this specific case, it is set to `MEMORY`, which means that the temporary table will be created in memory. If no engine is specified, the default storage engine will be used, which will store the table on disk.

Another option is to still declare the temporary table, but this time based on the results returned by a query.

```
17 • CREATE TEMPORARY TABLE current_season_races AS (
18       SELECT *
19       FROM race
20       WHERE date >= "2020-01-01"
21   );
```

Figure 130. Temporary Table from Query

If we now look at the fields of the temporary table by using the `SHOW FIELDS FROM` statement (Oracle Corporation, n.d. k), we see that the fields that were created are based on the results that were in the query.

```
25 • SHOW FIELDS FROM current_season_races;
```

Result Grid | Filter Rows: | Export: | Wrap

Field	Type	Null	Key	Default	Extra
race_id	int(11)	NO		0	NULL
racetrack_id	int(11)	NO		NULL	NULL
name	varchar(250)	NO		NULL	NULL
date	date	NO		NULL	NULL

Figure 131. Showing the Fields of the Temporary Table

And the data that is in the temporary table is the result set from the query used to create it. Including our shiny new record inserted via the stored procedure.

```
27 • SELECT *
28   FROM current_season_races;
```

Result Grid | Filter Rows: | Export:

race_id	racetrack_id	name	date
1	1	Austria 2020	2020-07-05
3	2	Hungary 2020	2020-07-19
6	3	New race via procedure	2020-08-02

Figure 132. Data in the Temporary Table

The temporary table can now be used just like a normal table in queries, while it exists.

The temporary table can be explicitly dropped using the `DROP TEMPORARY TABLE` statement. Otherwise, it will keep on existing until the connection is closed.

6 Recommended Additional Reading

Bradley, S., 2018. *The Types of Indexes You Can Add To MySQL Tables.* [Online] Available at: https://vanseodesign.com/web-design/the-types-of-indexes-you-can-add-to-mysql-tables/ [Accessed 11 December 2023].

Guru99, 2020c. *Clustered vs Non-clustered Index: Key Differences with Example.* [Online] Available at: https://www.guru99.com/clustered-vs-non-clustered-index.html [Accessed 11 December 2023].

mysqltutorial.com, 2020c. *MySQL Stored Procedures.* [Online] Available at: https://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx/ [Accessed 11 December 2023].

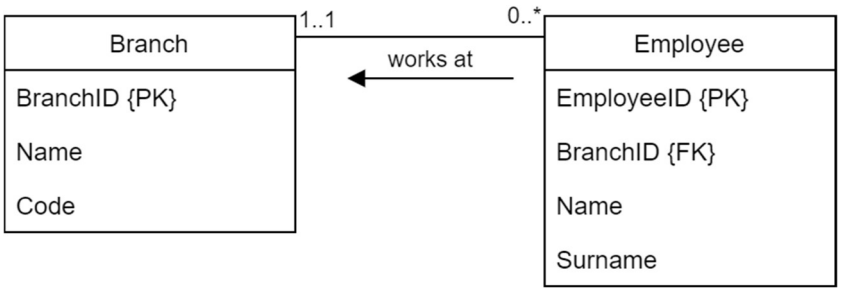
7 Revision Exercises

7.1 Revision Exercise 1

What is the difference between a view and a stored procedure?

7.2 Revision Exercise 2

Assume the below entity relationship diagram (ERD) has been implemented in a SQL database.



For which of the fields shown here will indexes be maintained?

8 Solutions to Revision Exercises

8.1 Revision Exercise 1

Read Sections 3.1 and 4.1 above.

8.2 Revision Exercise 1

Read Section 2 above.

Learning Unit 6: NoSQL Databases	
Learning Objectives: • Interpret JSON data. • Explain the benefits of using JSON. • Explain how to use MongoDB. • Write data to a NoSQL database. • Read data from a NoSQL database.	**My notes**
Material used for this learning unit: • This module manual. • MongoDB installation. • Visual Studio 2019 or a Java IDE.	
How to prepare for this learning unit: • Read through the content in this learning unit.	

1 Introduction

All the way back in Learning Unit 1, we saw that there are two major types of databases in use today: relational databases and NoSQL databases. The focus of this learning unit is learning how to use MongoDB – a widely used example of a NoSQL database. Let's see what MongoDB, Inc. has to say about their database.

> "**MongoDB** is a general purpose, document-based, distributed database built for modern application developers and for the cloud era." (MongoDB, Inc., 2020)

> "MongoDB is a **document database**, which means it stores data in JSON-like documents. We believe this is the most natural way to think about data and is much more expressive and powerful than the traditional row/column model." (MongoDB, Inc., 2020)

Think all the way back to Learning Unit 1, when we introduced the types of NoSQL databases. MongoDB is a document database–one of those four basic types that we mentioned. And as the second quote above indicates, it makes use of JSON-like documents to store data. So, let us first look at JSON before we jump into MongoDB.

2 JSON Documents

2.1 What is JSON?

Definition
JavaScript Object Notation (**JSON**) is “a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.” (json.org, n.d.)

There is quite a lot going on in this definition. The first part says that it is a “**lightweight data-interchange format.**” Systems exchange data with one another. There is, for example, data that is exchanged between the backend services of an e-commerce store, and the front-end code of the website that runs in the browser on your computer. There are web services provided by third parties, such as AccuWeather, that apps use to provide the user with up-to-date information. Data exchange is everywhere.

There are many different formats used to communicate the data. Some formats are proprietary binary standards. But for the most part, a few open standards are used when data is communicated over the internet. The first format that was popular was Extensible Markup Language (XML). It is also a human-readable format, where data is represented as a hierarchy of entities with attributes. Today, XML is considered quite verbose in comparison to JSON, and not used as often anymore.

That brings us to the **lightweight** part. JSON was specifically designed to be as concise and small enough to transmit as possible. Even though high-bandwidth connections are becoming more prevalent today, it is still wise to conserve the bandwidth that you do have. This is particularly true for servers. Imagine how much bandwidth is used by an online retailer if just 1 000 customers browse their catalogue at the same time! Then conserving every possible byte suddenly becomes quite important.

The next part of the definition talks about JSON being easy to read and write for **humans**. We will see in the next subsection how to construct JSON documents; it is indeed not hard to do.

And then the definition goes on to say that it is also easy for **machines** to generate and parse. This is, of course, actually the main purpose of JSON – for software applications to exchange data with one another.

There are many libraries that make working with JSON trivial. But even if you had to parse it quite manually, the structure is very simple to use.

2.2 Interpreting JSON Data

The information in this section is adapted from (The Independent Institute of Education, 2020, pp. 22-24).

In JSON, we can represent objects with properties, as well as arrays. Let's look at an example of a person with their name and age. This example was created in the online editor JSON Editor Online: https://jsoneditoronline.org/ [Accessed 11 December 2023].

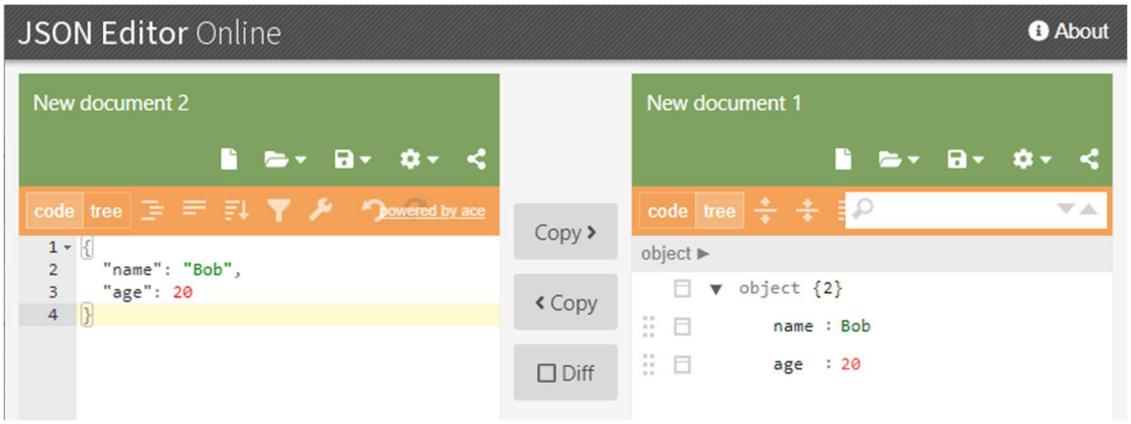


Figure 133. JSON Object

The curly brackets { } indicate the start and end of the object. And the properties are represented as key-value pairs. This is a very simple object, but objects could have many properties.

What if we wanted to represent another object inside of that, maybe containing information about the degree that Bob is studying? Well, objects can be nested, as shown in Figure 134.

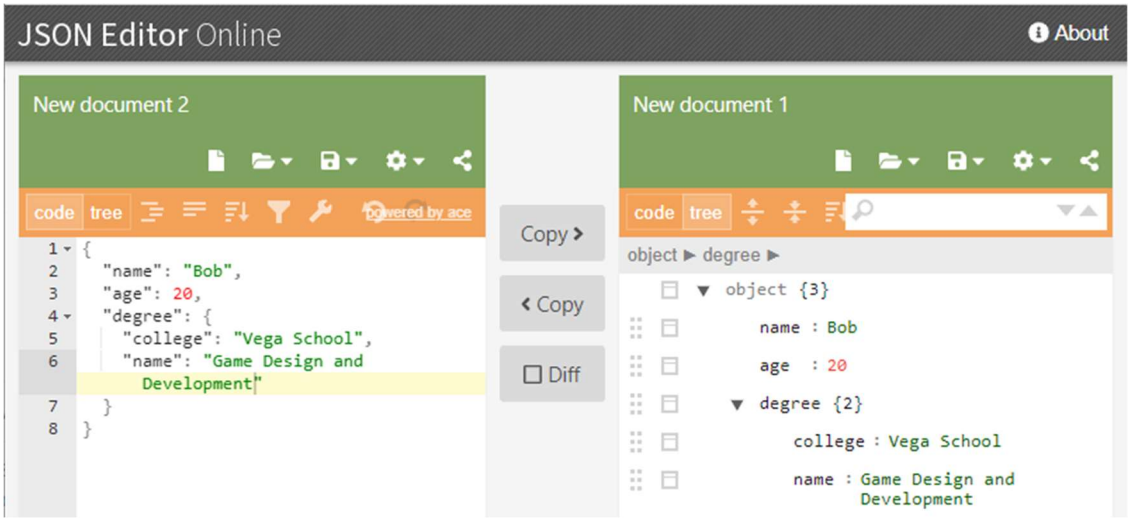


Figure 134. Nested Degree Object

One last thing that we want to be able to represent would be an array of objects. This is done using square brackets [ ], as shown in Figure 135.

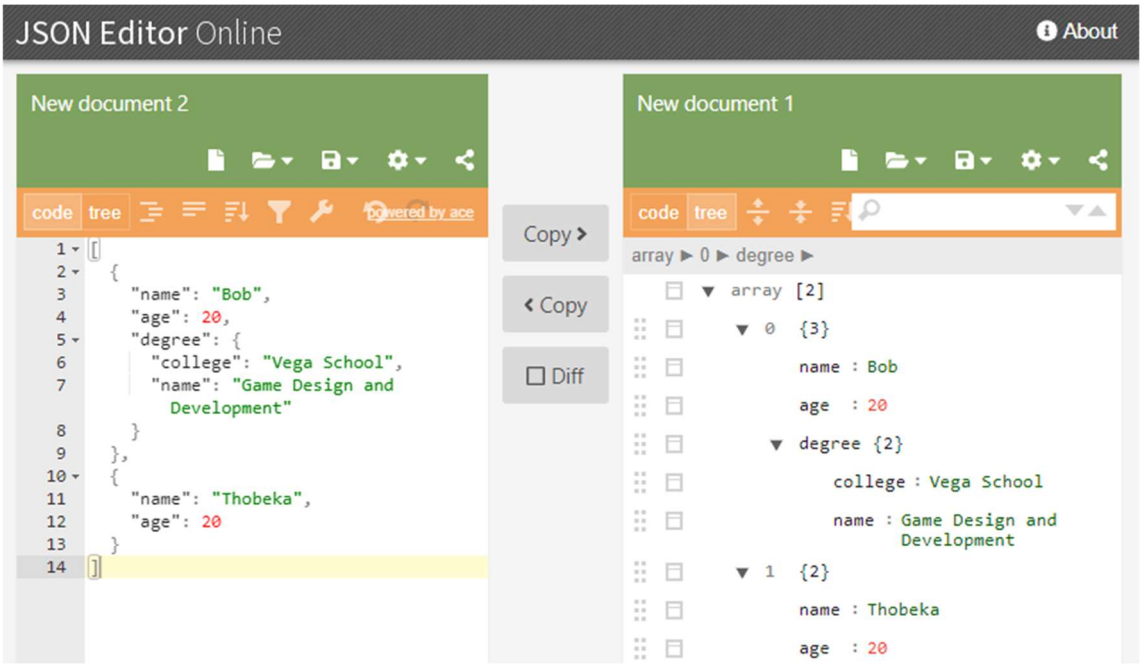


Figure 135. Array of Objects

Looking at this example, we see some of the flexibility of JSON. For the second person, no degree information is available. So, it is simply left out of the data representation.

By combining objects, properties, and arrays, we can represent any data using JSON.

Read more about JSON in (Tagliaferri, 2016).

3 Writing and Reading Data

3.1 Installing MongoDB

For this module, we are going to make use of **MongoDB Community Edition**. This can be installed on Windows, macOS, and a variety of Linux flavours.

Read the tutorial for your operating system from the list that appears here: https://docs.mongodb.com/manual/installation/ #mongodb-community-edition-installation-tutorials [Accessed 11 December 2023].

When prompted whether to install **MongoDB Compass**, do install that too. It is the graphical user interface (GUI) tools for the database.

If you don't want to install a local server, you can also make use of the **free tier** of the cloud hosted **MongoDB Atlas**, available here: https://www.mongodb.com/cloud/atlas [Accessed 11 December 2023].

3.2 Using MongoDB

There are two ways to access the data in MongoDB directly – using the command-line tools and using MongoDB Compass.

3.2.1 Command Line Tools

How to start up the command line tools for your operating system is described in the installation tutorial for that operating system. For Windows, run the following executable (MongoDB, Inc., n.d.):

C:\Program Files\MongoDB\Server\4.4\bin\mongo.exe

The following commands are useful when working with the command line tools (Jayatilake, 2019):

- `db` – tells you which database you are working with
- `show databases` – lists all the databases on the local server
- `use <database>` – replace the <database> part with the name of the database to either use (if it exists) or create (if it doesn't exist yet)
- `db.createCollection("myCollection")` – create a collection that can be any size
- `db.myCollection.insert()` – create a collection called myCollection if it doesn't exist yet, and insert the first document
- `db.myCollection.insertOne()` – inserts one document
- `db.myCollection.insertMany()` – inserts many documents
- `db.myCollection.find()` – queries and returns data
- `db.myCollection.update()` – updates existing data
- `db.myCollection.remove()` – deletes data from the database

The full list of commands can be found in the MongoDB documentation (MongoDB, 2021).

So, let us look an example of creating some of our Formula 1 data in MongoDB.

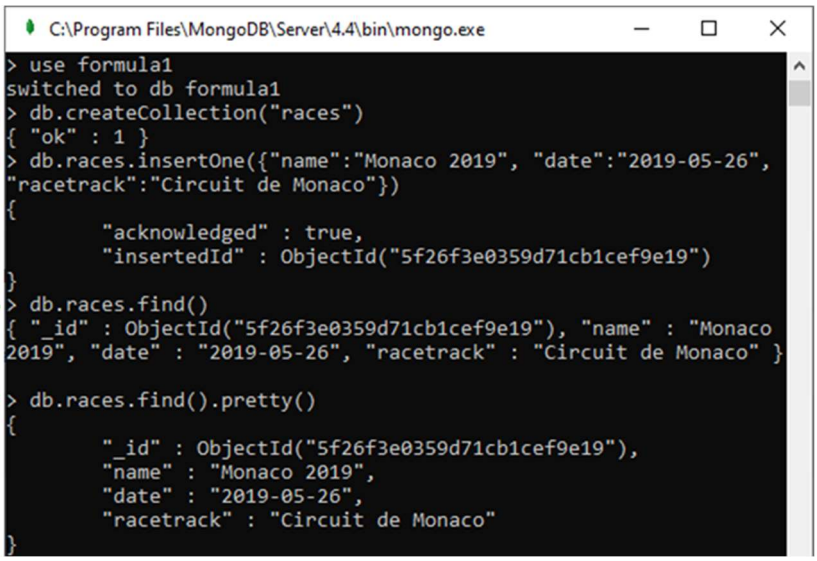


Figure 136. Inserting Data into MongoDB

Let us look at the commands line by line. Line 1 tells MongoDB that we are working with the `Formula1` database. If it does not exist yet, it gets created.

Line 2 creates a collection of documents called races.

Line 3 adds a document that describes the Monaco 2019 race. We use JSON syntax here to specify the data that we want to insert in key-value pairs. There are two things that you might notice at this point.

The first thing is that MongoDB automatically added an _id when we created the first document–the one that describes the Monaco 2019 race. The `insertedId` that is returned will be the assigned identifier for this document.

The other thing that is interesting is that something is absent. We didn't have to define any of the data structures. With a SQL database, you will remember that we spent a lot of time designing and creating tables. Here, we just jumped right in and created a document.

On line 4, we call the `find` command to retrieve all the documents that are in the `races` collection. The data is displayed in a valid JSON format but formatted in a way that isn't particularly reader-friendly. Which brings us to the final line of the example. Here we add the `pretty` command to display the same data but printed in an easier-to-read (for humans) format.

This is great, but not super exciting, with only one race in the database so far. Let us add another entry to work with. You can write the `insert` statement as an exercise. The data now looks like this:


```
> db.races.find().pretty()
{
        "_id" : ObjectId("5f26f3e0359d71cb1cef9e19"),
        "name" : "Monaco 2019",
        "date" : "2019-05-26",
        "racetrack" : "Circuit de Monaco"
}
{
        "_id" : ObjectId("5f26ffb56b3ee9e9f5843bfe"),
        "name" : "China 2019",
        "date" : "2019-04-14",
        "racetrack" : "Shanghai International Circuit"
}
```

Figure 137. A second entry

Now we can get into more interesting queries. Say we want to find the race with the name “China 2019”. As with everything in MongoDB, here we again provide the parameters that we want to match against as JSON data.

```
> db.races.find({"name": "China 2019"}).pretty()
{
        "_id" : ObjectId("5f26ffb56b3ee9e9f5843bfe"),
        "name" : "China 2019",
        "date" : "2019-04-14",
        "racetrack" : "Shanghai International Circuit"
}
```

Figure 138. Finding a race with a specific name

This is the equivalent of the where clause `WHERE name = “China 2019”` in SQL.

What if we wanted to find all the races with a date greater than 2019-05-01? We need a special syntax to represent that greater than: `$gt`. Less than would be represented with `$lt`.

```
> db.races.find({"date": { $gt: '2019-05-01'}}).pretty()
{
        "_id" : ObjectId("5f26f3e0359d71cb1cef9e19"),
        "name" : "Monaco 2019",
        "date" : "2019-05-26",
        "racetrack" : "Circuit de Monaco"
}
```

Figure 139. Greater than a date

This is great, but what if we wanted the date to be in a range, say between 1 April and 20 April? We can just add the second criterium by separating it with a comma.

```
> db.races.find({"date": { $gt: '2019-04-01', $lt: '2019-04-30'}}).pretty()
{
        "_id" : ObjectId("5f26ffb56b3ee9e9f5843bfe"),
        "name" : "China 2019",
        "date" : "2019-04-14",
        "racetrack" : "Shanghai International Circuit"
}
```

Figure 140. Date in a range

Another query we might want to write is finding all the races at racetracks starting with the letter S. This is equivalent to "`LIKE S%`" in SQL. This is a slightly more complex case. Here we need to make use of a regular expression. What is a regular expression?

Definition

"A regular expression is a method used in programming for pattern matching. Regular expressions provide a flexible and concise means to match strings of text." (Technopedia, 2011)

Different programming languages have slightly different variations on the syntax for regular expressions. MongoDB uses the Perl syntax for regular expressions. (MongoDB, 2021b)

Back to our current query – starting with the capital letter S.

```
> db.races.find({"racetrack": { $regex: /^S/ }}).pretty()
{
        "_id" : ObjectId("5f26ffb56b3ee9e9f5843bfe"),
        "name" : "China 2019",
        "date" : "2019-04-14",
        "racetrack" : "Shanghai International Circuit"
}
```

Figure 141. Starts with S

The `^` in the regular expression means that the `S` will match if it is at the start of the string. And matching something at the end of a string is marked by adding a `$` at the end of the regular expression. So, to find races that took place at racetracks ending in "Circuit", the expression is `/Circuit$/`.

```
> db.races.find({"racetrack": { $regex: /Circuit$/ }}).pretty()
{
        "_id" : ObjectId("5f26ffb56b3ee9e9f5843bfe"),
        "name" : "China 2019",
        "date" : "2019-04-14",
        "racetrack" : "Shanghai International Circuit"
}
```

Figure 142. Ends with Circuit

For a full explanation of everything that can be done with regular expressions in MongoDB, read (MongoDB, 2021b).

3.2.2 MongoDB Compass

As mentioned in the section about installing MongoDB, MongoDB Compass is a graphical user interface for interacting with the database. It is similar to MySQL Workbench and SQL Server Management Studio in functionality.

To connect MongoDB Compass to your local database:

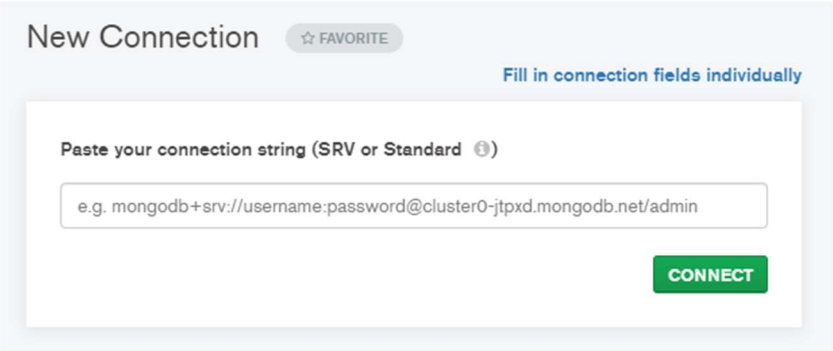


Figure 143. Filling in Fields Individually

1. Click **Fill in connection fields individually**.

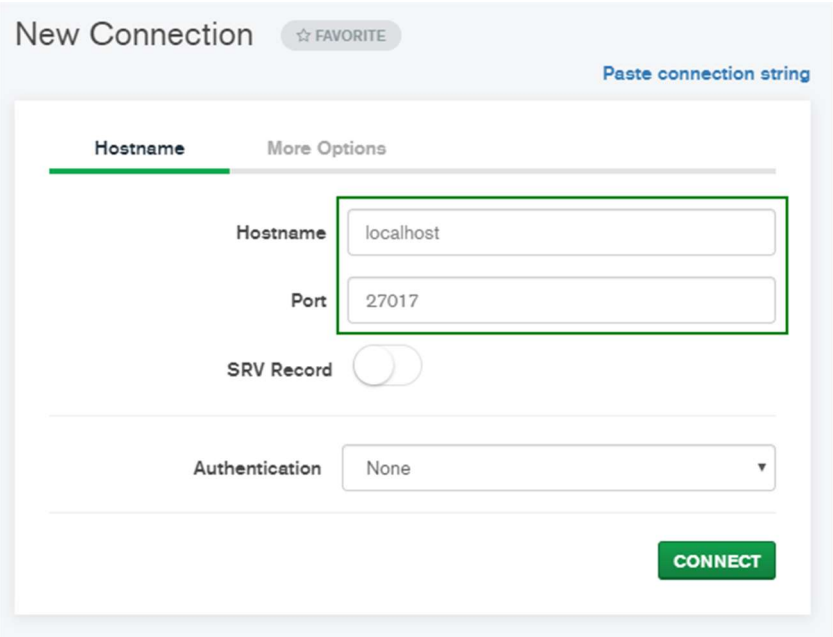


Figure 144. Check Connection Properties

2. The **Hostname** should be localhost and the **Port** 27017. (Liew, 2019)
3. Click **Connect**.

Then we can select the database and view the data that we created using the command line tools.

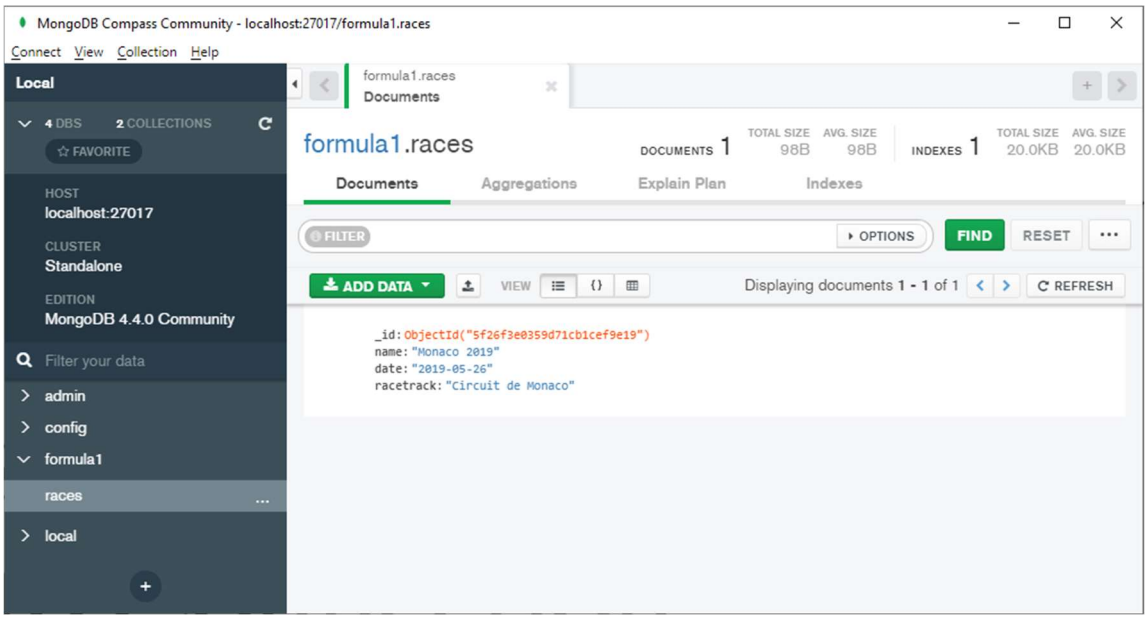


Figure 145. Viewing Data using MongoDB Compass

This user interface is quite intuitive to use. However, if you need more specific instructions, read the documentation available at: https://docs.mongodb.com/compass/master/ [Accessed 11 December 2023].

All this interacting with the database directly is quite fun for us as developers. But users of a system wouldn't be accessing the data in this way. They would be accessing a website or app, that would enable them to order pet food or upload a photo to social media. And that system would be accessing the data using a language like C# or Java. So, let's see how we can interface to MongoDB using these languages.

3.3 Writing and Reading Data in C#

In this section, we are going to use **Visual Studio 2019** to write C# code to access a MongoDB database. Read section 3.4 for information about writing and reading data in Java.

To connect to a MongoDB database using C#:

1. Start by creating a new **Console App (.NET Core)**.

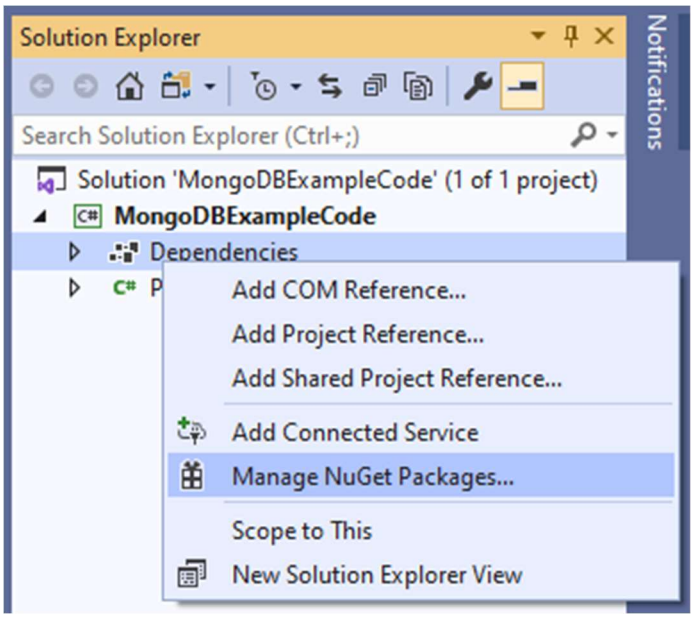


Figure 146. Manage NuGet Packages

2. Right-click on the **Dependencies** node and click **Manage NuGet Packages.**

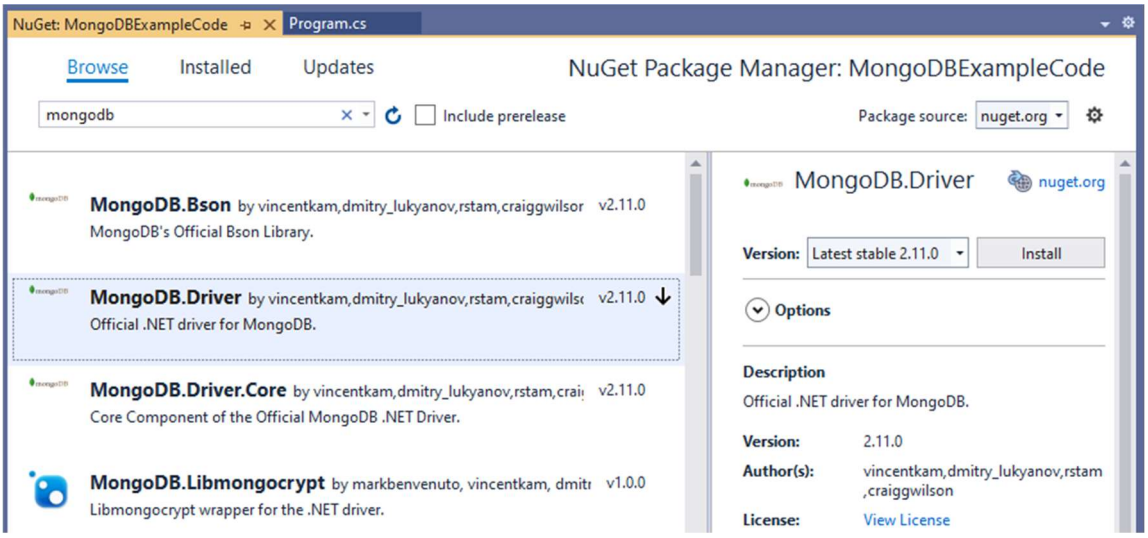


Figure 147. Searching for MongoDB

3. Click **Browse** (tab at the top) and search for *MongoDB*.
4. Select **MongoDB.Driver** and click **Install**.
5. Accept the prompts and wait for the process to complete.
6. Select **MongoDB.Bson** and **Install** that too.
7. Close the **NuGet** window.
8. Add the following code to the main method to write and read data. With thanks to (Alger, 2019), and (Alger, 2019b). Notice the two `using` statements at the top.


```
using MongoDB.Bson;
using MongoDB.Driver;
using System;

namespace MongoDBExampleCode
{
    class Program
    {
        static void Main(string[] args)
        {
            MongoClient = new MongoClient("mongodb://localhost:27017/");
            var database = mongoClient.GetDatabase("formula1");
            var collection = database.GetCollection<BsonDocument>("races");

            // add an item
            var document = new BsonDocument {
                { "name", "China 2019" },
                { "date", "2019-04-14" },
                { "racetrack", "Shanghai International Circuit"}
            };
            collection.InsertOne(document);

            // read all the items in the collection
            var items = collection.Find(new BsonDocument()).ToList();
            foreach (var item in items)
            {
                Console.WriteLine("Race:" + item.GetValue("name"));
                Console.WriteLine("Date: " + item.GetValue("date"));
                Console.WriteLine("Racetrack: " + item.GetValue("racetrack") +"\n");
            }
        }
    }
}
```

Figure 148 shows what the output of the console app looks like.

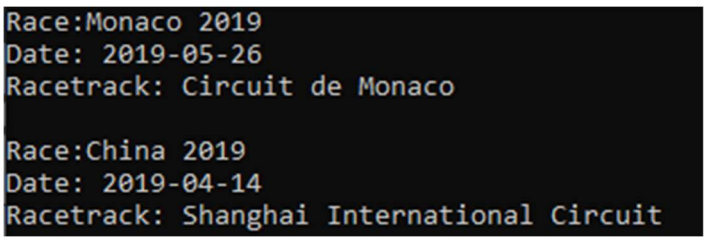


Figure 148. Output of the Console App

We can see that the C# application programming interface (API) works just like the command line one. So, if you have mastered that, using it in C# is easy.

3.4 Writing and Reading Data in Java

In this section, we are going to look at writing and reading data using Java. The steps described here are based on the information from (Fadatare, 2020) and (Kasyap, 2020) with modifications.

Use your favourite integrated development environment (IDE) for Java and create a Maven-based project. Include the following Maven dependency in the project (get the latest version from (MVNRepository, 2021):

```
<!-- https://mvnrepository.com/artifact/org.mongodb/mongo-java-driver -->
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongo-java-driver</artifactId>
    <version>3.12.10</version>
</dependency>
```


Add the following Java code to the main class to write and read data from the MongoDB database.

```
package za.ac.iie.insy6112.mongodb;

import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoCursor;
import org.bson.Document;

/**
 * Main class writing to and reading from MongoDB database.
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
         try (var mongoClient = MongoClients.create("mongodb://localhost:27017")) {

            var database = mongoClient.getDatabase("formula1");
            MongoCollection <Document> races = database.getCollection("races");

            // add to the collection
            Document = new Document();
            document.append("name", "Bahrain 2021");
            document.append("date", "2021-03-26");
            document.append("racetrack", "Bahrain International Circuit");
            races.insertOne(document);

            // read from the collection
            try (MongoCursor <Document> cur = races.find().iterator()) {
                while (cur.hasNext()) {
                    var race = cur.next();
                    System.out.println("Name: " + race.get("name", ""));
                    System.out.println("Date: " + race.get("date", ""));
                    System.out.println("Racetrack: " + race.get("racetrack", ""));
                    System.out.println();
                }
            }
        }
    }
}
```

Just like with the C# code, the Java code quite closely follows the command line commands.

4 Recommended Additional Reading

Work through the **MongoDB Tutorial for Beginners** by Chaitanya Singh. Available at: https://beginnersbook.com/ 2017/09/mongodb-tutorial/ [Accessed 11 December 2023].

Alger, K. W., 2019. *Quick Start: C# and MongoDB - Creating Documents.* [Online] Available at: https://www.mongodb.com/blog/post/quick-start-c-sharp-and-mongodb-creating-documents [Accessed 1 December 2023].

Alger, K. W., 2019b. *Quick Start: C# and MongoDB - Read Operations.* [Online] Available at: https://www.mongodb.com/blog/post/quick-start-c-and-mongodb-read-operations [Accessed 11 December 2023].

Liew, Z., 2019. *How to setup a local MongoDB Connection.* [Online] Available at: https://zellwk.com/blog/local-mongodb/ [Accessed 11 December 2023].

Singolia, V. 2021. *25 Most Common Commands for MongoDB Beginners*. [Online] Available at: https://www.codingninjas.com/ blog/2021/05/13/25-most-common-commands-for-mongodb-beginners/ [Accessed 11 December 2023].

Tagliaferri, L., 2016. *An Introduction to JSON.* [Online] Available at: https://www.digitalocean.com/community/tutorials/an-introduction-to-json [Accessed 11 December 2023].

5 Recommended Digital Engagement and Activities

Do some of the great free courses available at **MongoDB University**. Browse the list of available courses here: https://university.mongodb.com/courses/catalog [Accessed 11 December 2023].

6 Revision Exercises

6.1 Revision Exercise 1

How can a MongoDB be connected to from an application?

6.2 Revision Exercise 2

Write a MongoDB interactive shell command to insert the following data into the database:

Branch Code	Branch Name	Branch Type
163145	Brooklyn	Full service
153189	Somerset West	Tellers only

7 Solutions to Revision Exercises

7.1 Revision Exercise 1

Read Section 3.3 above.

7.2 Revision Exercise 2

Read Section 3.2.1 above.

Bibliography

Alger, K. W. 2019b. Quick Start: C# and MongoDB - Read Operations. [Online] Available at: https://www.mongodb.com/blog/post/quick-start-c-and-mongodb-read-operations [Accessed 11 December 2023].

Alger, K. W. 2019. Quick Start: C# and MongoDB - Creating Documents. [Online] Available at: https://www.mongodb.com/blog/post/quick-start-c-sharp-and-mongodb-creating-documents [Accessed 11 December 2023].

Amazon Web Services, Inc. 2020. What is a data lake?. [Online] Available at: https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/ [Accessed 8 December 2022].

Beck, A. 2013. File:SQL Joins.svg. [Online] Available at: https://commons.wikimedia.org/wiki/File:SQL_Joins.svg [Accessed 11 December 2023].

BitDegree. 2019. Correct Use of MySQL LIMIT OFFSET Clauses: Limiting and Paginating. [Online] Available at: https://www.bitdegree.org/learn/mysql-limit-offset [Accessed 11 December 2023].

Blaha, M. 2016. Be Careful with Derived Data. [Online] Available at: https://www.dataversity.net/careful-derived-data/ [Accessed 11 December 2023].

Bradley, S. 2018. The Types of Indexes You Can Add To MySQL Tables. [Online] Available at: https://vanseodesign.com/web-design/the-types-of-indexes-you-can-add-to-mysql-tables/ [Accessed 11 December 2023].

Carrington, D., 2020. How Many Photos Will Be Taken in 2020?. [Online] Available at: https://focus.mylio.com/tech-today/how-many-photos-will-be-taken-in-2020 [Accessed 11 December 2023].

Codd, E. 1970. A Relational Model of Data for Large Shared Databanks. [Online] Available at: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf [Accessed 11 December 2023].

Comfreak, n.d. Solar System Sun Mercury. [Online] Available at: https://pixabay.com/photos/solar-system-sun-mercury-venus-439046/ [Accessed 11 December 2023].

Coronel, C., Morris, S., Crockett, K. and Rob, P. 2013. Database Principles: Fundamentals of Design, Implementation and Management. 2nd ed. Andover: Cengage Learning EMEA.

Czenky, M. 2010. Normalization based on dependency. [Online] Available at: http://tmcs.math.unideb.hu/load_doc.php?p=188&t=doc [Accessed 11 December 2023].

Dalbey, J. 2005. Developing Entity Relationship Diagrams (ERDs). [Online] Available at: http://users.csc.calpoly.edu/~jdalbey/205/Lectures/HOWTO-ERD.html [Accessed 11 December 2023].

Data Entry Outsources. 2013. Components of a Database Management System. [Online] Available at: https://www.dataentryoutsourced.com/blog/components-of-a-database-management-system/ [Accessed 11 December 2023].

databasedev.co.uk, 2015. Relational Database Entity Integrity. [Online] Available at: http://www.databasedev.co.uk/entity_integrity.html [Accessed 11 December 2023].

Dybka, P. 2014b. Chen Notation. [Online] Available at: https://www.vertabelo.com/blog/chen-erd-notation/ [Accessed 8 December 2023].

Dybka, P. 2014c. UML Notation. [Online] Available at: https://www.vertabelo.com/blog/uml-notation/ [Accessed 11 December 2022].

Dybka, P. 2014. ERD Notations in Data Modeling. [Online] Available at: https://www.vertabelo.com/blog/comparison-of-erd-notations/ [Accessed 11 December 2023].

Dybka, P. 2016. Crow's Foot Notation. [Online] Available at: https://www.vertabelo.com/blog/crow-s-foot-notation/ [Accessed 11 December 2023].

Educative, Inc. 2020. What are ACID properties in a database?. [Online] Available at: https://www.educative.io/edpresso/what-are-acid-properties-in-a-database [Accessed 11 December 2023].

Elgabry, O. 2016. Database — Design Process (Part 3). [Online] Available at: https://medium.com/omarelgabrys-blog/database-design-process-part-3-7b5fafc78774 [Accessed 11 December 2023].

Encyclopædia Britannica, Inc., 2020. Declarative Languages. [Online] Available at: https://www.britannica.com/technology/computer-programming-language/Visual-Basic#ref248126 [Accessed 11 December 2023].

Fadatare, R. 2020. Java MongoDB Read Document Example. [Online] Available at: https://www.javaguides.net/2020/01/java-mongodb-read-document-example.html [Accessed 11 December 2023].

Fischer, P. n.d.. Formula 1 Pit Lane Ferrari. [Online] Available at: https://pixabay.com/photos/racing-formula-1-pit-lane-ferrari-3415413/ [Accessed 11 December 2023].

Foote, K. D. 2017. A Brief History of Data Modeling. [Online] Available at: https://www.dataversity.net/brief-history-data-modeling/ [Accessed 11 December 2023].

Foote, K. D. 2020. A Brief History of Database Management. [Online] Available at: https://www.dataversity.net/brief-history-database-management/# [Accessed 11 December 2023].

Formula One World Championship Limited, 2020. Hall of Fame - the World Champions. [Online] Available at: https://www.formula1.com/en/drivers/hall-of-fame.html [Accessed 11 December 2023].

Fowler, A. n.d.. Columnar Data in NoSQL. [Online] Available at: https://www.dummies.com/programming/big-data/columnar-data-in-nosql/ [Accessed 11 December 2023].

Fuller, A. 2006. Normalization: How far is far enough?. [Online] Available at: https://www.techrepublic.com/article/normalization-how-far-is-far-enough/ [Accessed 11 December 2023].

Garcia, R. 2019. 4 FREE ERD and diagram apps. [Online] Available at: https://medium.com/@ralphagarcia2017/4-free-erd-and-diagram-apps-f9b5cafb1110 [Accessed 11 December 2023].

Gartner, Inc. n.d.. Big Data. [Online] Available at: https://www.gartner.com/en/information-technology/glossary/big-data [Accessed 11 December 2023].

Gillingham, C. 2007. File:Modern Musical Notation.jpg. [Online] Available at: https://en.wikipedia.org/wiki/File:Modern_Musical_Notation.jpg [Accessed 11 December 2023].

Google Scholar. 2020. databases - Google Scholar. [Online] Available at: https://scholar.google.com/scholar?as_ylo=2020&q=databases&hl=en&as_sdt=0,5 [Accessed 11 December 2023].

Guru99. 2020b. ER Diagram Tutorial in DBMS (with Example). [Online] Available at: https://www.guru99.com/er-diagram-tutorial-dbms.html [Accessed 8 December 2022].

Guru99. 2020c. Clustered vs Non-clustered Index: Key Differences with Example. [Online] Available at: https://www.guru99.com/clustered-vs-non-clustered-index.html [Accessed 11 December 2023].

Guru99. 2020. Waterfall Vs. Agile: Must Know Differences. [Online] Available at: https://www.guru99.com/waterfall-vs-agile.html [Accessed 8 December 2022].

Guyer, C. et al., 2017. Views. [Online] Available at: https://docs.microsoft.com/en-us/sql/relational-databases/views/views?view=sql-server-ver15 [Accessed 8 December 2022].

Guyer, C. et al. 2019. Clustered and Nonclustered Indexes Described. [Online] Available at: https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver15 [Accessed 8 December 2022].

Hijleh, M. A. 2016. File:BigDataVs.png. [Online] Available at: https://commons.wikimedia.org/wiki/File:BigDataVs.png [Accessed 8 December 2022].

Holowczak, R. n.d.. Drawing Entity Relationship Diagrams with UML Notation using LucidChart. [Online] Available at: http://holowczak.com/drawing-entity-relationship-diagrams-with-uml-notation-using-lucidchart/ [Accessed 8 December 2022].

Howarth, F. 2014. 5 Key Steps to Ensuring Database Security. [Online] Available at: https://www.dbta.com/Editorial/Think-About-It/5-Key-Steps-to-Ensuring-Database-Security-95307.aspx [Accessed 8 December 2022].

IIE Library. 2020. IIE Library Search Results. [Online] Available at: https://libraryconnect.iie.ac.za/client/en_GB/iie/search/results?qu=database+principles&te= [Accessed 11 December 2023].

Import.io. 2019. What is Data Normalization and Why Is It Important?. [Online] Available at: https://www.import.io/post/what-is-data-normalization-and-why-is-it-important/ [Accessed 11 December 2023].

JavaTpoint, n.d.. MySQL Temporary Table. [Online] Available at: https://www.javatpoint.com/mysql-temporary-table [Accessed 11 December 2023].

Jayatilake, N. 2019. How to get started with MongoDB in 10 minutes. [Online] Available at: https://www.freecodecamp.org/news/learn-mongodb-a4ce205e7739/ [Accessed 11 December 2023].

Johnson, P. 2016. When to Consider a NoSQL vs Relational Database. [Online] Available at: https://resources.whitesourcesoftware.com/blog-whitesource/when-to-consider-a-nosql-vs-relational-database [Accessed 11 December 2023].

Jorrit. 2011. File:ER Diagram MMORPG.svg. [Online] Available at: https://commons.wikimedia.org/wiki/File:ER_Diagram_MMORPG.svg [Accessed 11 December 2023].

Joshi, N. 2017. Top 5 sources of big data. [Online] Available at: https://www.allerin.com/blog/top-5-sources-of-big-data [Accessed 11 December 2023].

json.org, n.d. Introducing JSON. [Online] Available at: https://www.json.org/json-en.html [Accessed 11 December 2023].

Kalodikis, C. 2016. Data Dictionary (Database). [Online] Available at: https://www.youtube.com/watch?v=kH0bcw9P2Lc [Accessed 8 December 2022].

Kasyap, K. 2020. How to insert a document into a MongoDB collection using Java?. [Online] Available at: https://www.tutorialspoint.com/how-to-insert-a-document-into-a-mongodb-collection-using-java [Accessed 11 December 2023].

Kauffman, L. 2013. About Secure Password Hashing. [Online] Available at: https://security.blogoverflow.com/2013/09/about-secure-password-hashing/ [Accessed 11 December 2023].

Khalil, M. 2018. SQL Server, PostgreSQL, MySQL... what's the difference? Where do I start?. [Online] Available at: https://www.datacamp.com/community/blog/sql-differences [Accessed 11 December 2023].

Kirs, P. J. 2003. What is an ERD?. [Online] Available at: http://www.pkirs.utep.edu/cis4365/Tutorials/ERD/tutorial%203.00010/ERDs.htm [Accessed 11 December 2023].

Knight, M. 2017. What is a Data Dictionary?. [Online] Available at: https://www.dataversity.net/what-is-a-data-dictionary/ [Accessed 11 December 2023].

Larsen, G. A. 2011. SQL Server: Natural Key Verses Surrogate Key. [Online] Available at: https://www.databasejournal.com/features/mssql/article.php/3922066/SQL-Server-Natural-Key-Verses-Surrogate-Key.htm [Accessed 11 December 2023].

Liew, Z. 2019. How to setup a local MongoDB Connection. [Online] Available at: https://zellwk.com/blog/local-mongodb/ [Accessed 11 December 2023].

Lucid Software Inc. 2020. Entity-Relationship Diagram Symbols and Notation. [Online] Available at: https://www.lucidchart.com/pages/ER-diagram-symbols-and-meaning [Accessed 11 December 2023].

Marr, B. 2019. What's The Difference Between Structured, Semi-Structured And Unstructured Data?. [Online] Available at: https://www.forbes.com/sites/bernardmarr/2019/10/18/whats-the-difference-between-structured-semi-structured-and-unstructured-data/ [Accessed 11 December 2023].

MathsIsFun.com. 2018. Definition of Notation. [Online] Available at: https://www.mathsisfun.com/definitions/notation.html [Accessed 11 December 2023].

Merriam-Webster, n.d. Definition of Noun. [Online] Available at: https://www.merriam-webster.com/dictionary/noun [Accessed 11 December 2023].

Microsoft. 2020. Data Lake. [Online] Available at: https://azure.microsoft.com/en-us/solutions/data-lake/ [Accessed 11 December 2023].

Milener, G., Guyer, C., Rohm, W. A. and Hamilton, B. 2020. CREATE VIEW (Transact-SQL). [Online] Available at: https://docs.microsoft.com/en-us/sql/t-sql/statements/create-view-transact-sql?view=sql-server-ver15 [Accessed 11 December 2023].

Monge, A., n.d. Basic structures: Database Relationships. [Online] Available at: https://web.csulb.edu/colleges/coe/cecs/dbdesign/dbdesign.php?page=association.php [Accessed 11 December 2023].

MongoDB, Inc. 2020. The database for modern applications. [Online] Available at: https://www.mongodb.com/ [Accessed 11 December 2023].

MongoDB, Inc., n.d.. Install MongoDB Community Edition on Windows. [Online] Available at: https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/ [Accessed 11 December 2023].

MongoDB. 2021b. $regex. [Online] Available at: https://docs.mongodb.com/manual/reference/operator/query/regex/ [Accessed 11 December 2023].

MongoDB. 2021. Database Commands. [Online] Available at: https://docs.mongodb.com/manual/reference/command/ [Accessed 11 December 2023].

MVNRepository. 2021. MongoDB Java Driver. [Online] Available at: https://mvnrepository.com/artifact/org.mongodb/mongo-java-driver [Accessed 11 December 2023].

mysqltutorial.com. 2020c. MySQL Stored Procedures. [Online] Available at: https://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx/ [Accessed 11 December 2023].

mysqltutorial.org. 2020b. Introduction to MySQL Stored Procedures. [Online] Available at: https://www.mysqltutorial.org/introduction-to-sql-stored-procedures.aspx [Accessed 11 December 2023].

mysqltutorial.org. 2020. MySQL LIKE. [Online] Available at: https://www.mysqltutorial.org/mysql-like/ [Accessed 11 December 2023].

mysqltutorial.org, n.d.. MySQL Transaction. [Online] Available at: https://www.mysqltutorial.org/mysql-transaction.aspx/ [Accessed 11 December 2023].

Object Management Group, n.d. Data Modeling Methodology and Notation. [Online] Available at: https://www.omg.org/retail-depository/arts-odm-73/data_modeling_methodology_and_.htm [Accessed 8 December 2022].

Olamendy, J. C. 2017. Temporary Tables in MySQL. [Online] Available at: https://blog.toadworld.com/2017/09/27/temporary-tables-in-mysql [Accessed 8 December 2022].

OmniSci, Inc. 2019. DBMS Definition. [Online] Available at: https://www.omnisci.com/technical-glossary/dbms [Accessed 8 December 2022].

Oracle Corporation, n.d. c. 13.1.12 CREATE DATABASE Statement. [Online] Available at: https://dev.mysql.com/doc/refman/8.0/en/create-database.html [Accessed 11 December 2023].

Oracle Corporation, n.d. d. 9.2.3 Identifier Case Sensitivity. [Online] Available at: https://dev.mysql.com/doc/refman/8.0/en/identifier-case-sensitivity.html [Accessed 11 December 2023].

Oracle Corporation, n.d. e. 11.3.1 String Data Type Syntax. [Online] Available at: https://beginnersbook.com/2015/04/instance-and-schema-in-dbms/ [Accessed 8 December 2022].

Oracle Corporation, n.d. f. 10.2 Character Sets and Collations in MySQL. [Online] Available at: https://dev.mysql.com/doc/refman/8.0/en/charset-mysql.html [Accessed 11 December 2023].

Oracle Corporation, n.d. g. 11.2.2 The DATE, DATETIME, and TIMESTAMP Types. [Online] Available at: https://dev.mysql.com/doc/refman/8.0/en/datetime.html [Accessed 11 December 2023].

Oracle Corporation, n.d. g. 12.4.2 Comparison Functions and Operators. [Online] Available at: https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html [Accessed 11 December 2023].

Oracle Corporation, n.d. h. 12.20.1 Aggregate Function Descriptions. [Online] Available at: https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html [Accessed 11 December 2023].

Oracle Corporation, n.d. i. 8.3.1 How MySQL Uses Indexes. [Online] Available at: https://dev.mysql.com/doc/refman/8.0/en/mysql-indexes.html [Accessed 11 December 2023].

Oracle Corporation, n.d. j. 6.1.5 Working with Stored Procedures. [Online] Available at: https://dev.mysql.com/doc/connector-net/en/connector-net-tutorials-stored-procedures.html [Accessed 11 December 2023].

Oracle Corporation, n.d. k. 13.7.7.5 SHOW COLUMNS Statement. [Online] Available at: https://dev.mysql.com/doc/refman/8.0/en/show-columns.html [Accessed 11 December 2023].

Oracle Corporation, n.d.b. Chapter 13 SQL Statements. [Online] Available at: https://dev.mysql.com/doc/refman/8.0/en/sql-statements.html [Accessed 11 December 2023].

Oracle Corporation, n.d.. Chapter 14 MySQL Data Dictionary. [Online] Available at: https://dev.mysql.com/doc/refman/8.0/en/data-dictionary.html [Accessed 11 December 2023].

Oracle, Inc. 2020b. The Definition of Big Data. [Online] Available at: https://www.oracle.com/za/big-data/what-is-big-data.html [Accessed 11 December 2023].

Oracle, 2020. What Is a Database?. [Online] Available at: https://www.oracle.com/za/database/what-is-database.html [Accessed 11 December 2023].

Oracle, n.d. Password Hashing. [Online] Available at: https://docs.oracle.com/cd/E26180_01/Platform.94/ATGPersProgGuide/html/s0506passwordhashing01.html [Accessed 11 December 2023].

Parahar, M. 2019. Difference between DDL and DML in DBMS.. [Online] Available at: https://www.tutorialspoint.com/difference-between-ddl-and-dml-in-dbms [Accessed 11 December 2023].

PCMag Digital Group. 2020. Definition of Primary Key. [Online] Available at: https://www.pcmag.com/encyclopedia/term/primary-key [Accessed 11 December 2023].

Pearson Education. 2009. Chapter 12: Entity-Relationship Modeling. [Online] Available at: http://web.eecs.utk.edu/~bvanderz/cs465/notes/ch12.pdf [Accessed 11 December 2023.

Pellissier, H. 2019. The Process - A Brief Description. [Online] Available at: https://www.pellissier.co.za/wp-content/uploads/erds/content/index.html#/lessons/l2aG_0RrVcYOIg5p6FyhM4g6pg0hNJGZ [Accessed 11 December 2023].

Plew, R. and Stephens, R., 2003. The Database Normalization Process. [Online] Available at: https://www.informit.com/articles/article.aspx?p=30646 [Accessed 11 December 2023].

Rouse, M. 2014. data abstraction. [Online] Available at: https://whatis.techtarget.com/definition/data-abstraction [Accessed 11 December 2023].

Rouse, M. 2020. data dictionary. [Online] Available at: https://searchapparchitecture.techtarget.com/definition/data-dictionary [Accessed 11 December 2023].

Sezairi, A. 2019. What is an Entity-Relationship Diagram?. [Online] Available at: https://medium.com/better-programming/what-is-an-entity-relationship-diagram-d5db69a87971 [Accessed 11 December 2023].

Shay, T. 2018. SQL Order of Operations – In Which Order MySQL Executes Queries?. [Online] Available at: https://www.eversql.com/sql-order-of-operations-sql-query-order-of-execution/ [Accessed 11 December 2023].

Simpson, K. 2016. The Difference Between Operational and Analytical Data Systems. [Online] Available at: https://www.arkatechture.com/blog/the-difference-between-operational-and-analytical-data-systems [Accessed 11 December 2023].

Soderberg, R. 2019. Comparing SQL Views and Stored Procedures. [Online] Available at: https://dev.to/rachelsoderberg/comparing-sql-views-and-stored-procedures-4pfb [Accessed 11 December 2023].

South African Government. 2013. Government Gazette, 26 November 2013. [Online] Available at: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.gov.za/sites/default/files/gcis_document/201409/3706726-11act4of2013protectionofpersonalinforcorrect.pdf [Accessed 8 December 2023].

SQLBolt, 2019. SQL Lesson 12: Order of execution of a Query. [Online] Available at: https://sqlbolt.com/lesson/select_queries_order_of_execution [Accessed 11 December 2023].

SQLite.org, n.d. About SQLite. [Online] Available at: https://www.sqlite.org/about.html [Accessed 11 December 2023].

Studytonight. 2020. Components of DBMS. [Online] Available at: https://www.studytonight.com/dbms/components-of-dbms.php [Accessed 11 December 2023].

Sybase Inc. 2003. Benefits of normalization. [Online] Available at: http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.dc20020_1251/html/databases/databases216.htm [Accessed 11 December 2023].

Tagliaferri, L. 2016. An Introduction to JSON. [Online] Available at: https://www.digitalocean.com/community/tutorials/an-introduction-to-json [Accessed 11 December 2023].

Tang, D. 2012. CS241 -- Lecture Notes: B-Trees. [Online] Available at: https://www.cpp.edu/~ftang/courses/CS241/notes/b-tree.htm [Accessed 11 December 2023].

Technopedia. 2011. What Does Regular Expression Mean?. [Online] Available at: https://www.techopedia.com/definition/25843/regular-expression [Accessed 11 December 2023].

TechOnTheNet.com, n.d. MySQL: EXISTS Condition. [Online] Available at: https://www.techonthenet.com/mysql/exists.php [Accessed 8 December 2022].

The Independent Institute of Education, 2020. Open Source Coding (Intermediate) Module Manual 2020. 1st ed. Sandton: The Independent Institute of Education.
tutorialspoint.com, 2020. MySQL - Between Clause. [Online] Available at: https://www.tutorialspoint.com/mysql/mysql-between-clause.htm [Accessed 11 December 2023].

University of Cape Town. 2017. Chapter 6. Entity-Relationship Modelling. [Online] Available at: https://www.cs.uct.ac.za/mit_notes/database/htmls/chp06.html [Accessed 11 December 2023].

Upadhyay, M. n.d.. First Normal Form (1NF). [Online] Available at: https://www.geeksforgeeks.org/first-normal-form-1nf/ [Accessed 11 December 2023].

Vettor, R. et al. 2020. Relational vs. NoSQL data. [Online] Available at: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/relational-vs-nosql-data [Accessed 11 December 2023].

Watt, A. and Eng, N., 2014. Database Design. 2nd edition.. [Online] Available at: https://open.bccampus.ca/browse-our-collection/find-open-textbooks/?uuid=5b6f010a-0563-44d4-94c5-67caa515d2c5 [Accessed 11 December 2023].

www.popiact-compliance.co.za, n.d. POPI Act Compliance. [Online] Available at: https://www.popiact-compliance.co.za/popia-information [Accessed 11 December 2023].

www.softwaretestinghelp.com. 2020. Database Normalization Tutorial: 1NF 2NF 3NF BCNF Examples. [Online] Available at: https://www.softwaretestinghelp.com/database-normalization-tutorial/ [Accessed 11 December 2023].

Intellectual Property

Plagiarism occurs in a variety of forms. Ultimately though, it refers to the use of the words, ideas or images of another person without acknowledging the source using the required conventions. The IIE publishes a Quick Reference Guide that provides more detailed guidance, but a brief description of plagiarism and referencing is included below for your reference. It is vital that you are familiar with this information and the Intellectual Integrity Policy before attempting any assignments.

Introduction to Referencing and Plagiarism

What is 'Plagiarism'?

'Plagiarism' is the act of taking someone's words or ideas and presenting them as your own.

What is 'Referencing'?

'Referencing' is the act of citing or giving credit to the authors of any work that you have referred to or consulted. A 'reference' then refers to a citation (a credit) or the actual information from a publication that is referred to.

Referencing is the acknowledgment of any work that is not your own, but is used by you in an academic document. It is simply a way of giving credit to and acknowledging the ideas and words of others.

When writing assignments, students are required to acknowledge the work, words or ideas of others through the technique of referencing. Referencing occurs in the text at the place where the work of others is being cited, and at the end of the document, in the bibliography.

The bibliography is a list of all the work (published and unpublished) that a writer has read in the course of preparing a piece of writing. This includes items that are not directly cited in the work.

A reference is required when you:

- Quote directly: when you use the exact words as they appear in the source;
- Copy directly: when you copy data, figures, tables, images, music, videos or frameworks;
- Summarise: when you write a short account of what is in the source;
- Paraphrase: when you state the work, words and ideas of someone else in your own words.

It is standard practice in the academic world to recognise and respect the ownership of ideas, known as intellectual property, through good referencing techniques. However, there are other reasons why referencing is useful.

Good Reasons for Referencing

It is good academic practice to reference because:

- It enhances the quality of your writing;
- It demonstrates the scope, depth and breadth of your research;
- It gives structure and strength to the aims of your article or paper;
- It endorses your arguments;
- It allows readers to access source documents relating to your work, quickly and easily.

Sources

The following would count as 'sources':

- Books,
- Chapters from books,
- Encyclopaedias,
- Articles,
- Journals,
- Magazines,
- Periodicals,
- Newspaper articles,
- Items from the Internet (images, videos, etc.),
- Pictures,
- Unpublished notes, articles, papers, books, manuscripts, dissertations, theses, etc.,
- Diagrams,
- Videos,
- Films,
- Music,
- Works of fiction (novels, short stories or poetry).

What You Need to Document from the Hard Copy Source You are Using

(Not every detail will be applicable in every case. However, the following lists provide a guide to what information is needed.)

You need to acknowledge:

- The words or work of the author(s),
- The author(s)'s or editor(s)'s full names,
- If your source is a group/ organisation/ body, you need all the details,
- Name of the journal, periodical, magazine, book, etc.,
- Edition,
- Publisher's name,
- Place of publication (i.e. the city of publication),
- Year of publication,
- Volume number,
- Issue number,
- Page numbers.

What You Need to Document if you are Citing Electronic Sources

- Author(s)'s/ editor(s)'s name,
- Title of the page,
- Title of the site,
- Copyright date, or the date that the page was last updated,
- Full Internet address of page(s),
- Date you accessed/ viewed the source,
- Any other relevant information pertaining to the web page or website.

Referencing Systems

There are a number of referencing systems in use and each has its own consistent rules. While these may differ from system-to-system, the referencing system followed needs to be used consistently, throughout the text. Different referencing systems cannot be mixed in the same piece of work!

A detailed guide to referencing, entitled Referencing and Plagiarism Guide is available from your library. Please refer to it if you require further assistance.

When is Referencing Not Necessary?

This is a difficult question to answer – usually when something is 'common knowledge'. However, it is not always clear what 'common knowledge' is.

Examples of ‘common knowledge’ are:

- Nelson Mandela was released from prison in 1990;
- The world’s largest diamond was found in South Africa;
- South Africa is divided into nine (9) provinces;
- The lion is also known as ‘The King of the Jungle’.
- $E = mc^2$
- The sky is blue.

Usually, all of the above examples would not be referenced. The equation $E = mc^2$ is Einstein’s famous equation for calculations of total energy and has become so familiar that it is not referenced to Einstein.

Sometimes what we think is ‘common knowledge’, is not. For example, the above statement about the sky being blue is only partly true. The light from the sun looks white, but it is actually made up of all the colours of the rainbow. Sunlight reaches the Earth's atmosphere and is scattered in all directions by all the gases and particles in the air. The smallest particles are by coincidence the same length as the wavelength of blue light. Blue is scattered more than the other colours because it travels as shorter, smaller waves. It is not entirely accurate then to claim that the sky is blue. It is thus generally safer to always check your facts and try to find a reputable source for your claim.

Important Plagiarism Reminders

The IIE respects the intellectual property of other people and requires its students to be familiar with the necessary referencing conventions. Please ensure that you seek assistance in this regard before submitting work if you are uncertain.

If you fail to acknowledge the work or ideas of others or do so inadequately this will be handled in terms of the Intellectual Integrity Policy (available in the library) and/ or the Student Code of Conduct – depending on whether or not plagiarism and/ or cheating (passing off the work of other people as your own by copying the work of other students or copying off the Internet or from another source) is suspected.

Your campus offers individual and group training on referencing conventions – please speak to your librarian or ADC/ Campus Co-Navigator in this regard.

Reiteration of the Declaration you have signed:

1. I have been informed about the seriousness of acts of plagiarism.
2. I understand what plagiarism is.
3. I am aware that The Independent Institute of Education (IIE) has a policy regarding plagiarism and that it does not accept acts of plagiarism.
4. I am aware that the Intellectual Integrity Policy and the Student Code of Conduct prescribe the consequences of plagiarism.

5. I am aware that referencing guides are available in my student handbook or equivalent and in the library and that following them is a requirement for successful completion of my programme.
6. I am aware that should I require support or assistance in using referencing guides to avoid plagiarism I may speak to the lecturers, the librarian or the campus ADC/ Campus Co-Navigator.
7. I am aware of the consequences of plagiarism.

Please ask for assistance prior to submitting work if you are at all unsure.